OBJECT-ORIENTED MODELING BASED ON INFORMATION GRAMMARS

een wetenschappelijke proeve op het gebied
van de
Wiskunde en Informatica

Proefschrift

ter verkrijging van de graad van doctor
aan de Katholieke Universiteit Nijmegen,
volgens besluit van het College van Decanen
in het openbaar te verdedigen op
maandag 9 juni 1997
des namiddags om **1.30 uur** precies

door

Paulus Johannes Maria Frederiks

geboren op 30 april 1968 te Nijmegen

Promotor:	Prof. C.H.A. Koster
Co-promotor:	Dr. ir. Th.P. van der Weide

Manuscriptcommissie:

Prof. dr. W. Gerhardt	Technische Universiteit Delft
Dr. H.A. Proper	Queensland University of Technology, Australia
Prof. dr. R.P. van de Riet	Vrije Universiteit Amsterdam

ISBN 90-9010338-4
NUGI 851
1997

Contents

Preface

There 've been good times
There 've been bad times

From: "Good Times, Bad Times",
The Rolling Stones

Writing a PhD thesis is a job which can not be done without moral support and support with respect to content. At this place I would like to thank all people who have supported me in any way.

I would like to thank my promotor Kees Koster for his involvement with my PhD project. We have had many discussions on the subject of object-orientation, context-free grammars, programming languages and conceptual modeling. His experience with, and overview of, computer science in general has been very helpful to me and has helped me to look at our research in a wider perspective. Furthermore, I would like to thank him for reading our reports and providing these reports with useful and constructive criticism.

In the first two years of my career as a PhD student, Arthur ter Hofstede was my supervisor. He spared no pains to introduce me in the fields of data bases, information systems and all kinds of formalisms. Category theory was such a formalism. Together with Ernst Lippe, a former colleague of Arthur, we have written two articles on the boundary line of conceptual data modeling and category theory. Thanks to Arthur's experience in writing and publishing papers these two articles have been published in scientific journals. His knowledge about formalisms, conceptual modeling, and his passion for scientific research have made a deep impression on me.

For the second half of my PhD period, Theo van der Weide became my supervisor. At that time I was having a rough period. Theo pulled me through this period and learned me to believe in myself, to be more diplomatic and to act resolutely. In these two years he put a lot of effort and time in my (our) research, and was a good listener to my problems and doubts. He inspired and encouraged me to write papers, to give lectures and to present our research results to people of the business community as well as colleague researchers. In this period we worked intensively together which has resulted in many papers. Besides working in his room at the university we often worked at his home. For this I would also like to thank his wife Nell Coumans and their family. Theo, thanks very much!

I would also like to thank my colleagues of the research priority Software Engineering

and our research line Information Systems. Especially, I would like to thank Frits Berger, Patrick van Bommel, Rob Bosman, Caspar Derksen, Franc Grootjen, Paul Jones, Ineke Kuster and Greta Löw. With Frits I shared a room at the university and a same path in our PhD projects. We have had extremely much fun together and we always found new things to joke about. Every now and then we played a couple of games called stratego. Together with Patrick we discussed and laughed in the pub. Furthermore, Patrick shared his experience of writing a PhD thesis with me. I would like to thank him for his advice and encouragements. Rob always looks at the bright side of life. It was always amusing to hear him talk about good food and wine, his bargains and his passion for Ajax. Together with his roommates Caspar, Franc and Paul there was always time for a good laugh. Our secretaries Greta and Ineke took a lot of administrative work out off my hands and they gave me also a lot of moral support. Furthermore, I would like to thank my Master's student Erik van de Ven. It was very pleasant to support and work with an enthusiastic person such as Erik.

As stated before, doing research is not a solo performance. With Patrick van Bommel, Caspar Derksen, Arthur ter Hofstede, Kees Koster, Ernst Lippe, Erik van de Ven and Theo van der Weide I have written one or more papers. Their contributions to this thesis are herewith gratefully acknowledged. Furthermore, I like to thank all people who read preliminary versions of this thesis. The constructive criticism of Caspar Derksen and Kees Koster have lead to several improvements of this thesis.

Besides my colleagues at the university I would like to thank my friends and the members of the rock 'n' roll bands I play (played) in. With Freek Engels, Herman Honer, Halbe Huitema, Arjan Knijff, Reinoud van der Korst, Theo Rouschop and many other friends I have had many pleasant moments and good talks during the last four years. I would also like to thank The Rolling Stones for their records, which always have been inspiring to me. For this reason each chapter of this thesis starts with a quote from a song of The Rolling Stones.

Furthermore, I want to thank my family. My mother has always been a good listener. She has always believed in me and she is always there for me. My father gave me (and my wife) a lot of good advice and moral support during the last four years. He also gave me a piece of ground within his garden where I could relax and escape for a moment from my research activities. I am very happy to see them both in the garden often.

Finally, I would like to thank my wife Marsha Tjon A Tjieuw. Although she had her own problems in finding a proper job she was always there when I needed her. She encouraged me to continue and finish this job in good times and in bad times. Therefore, I would like to dedicate this thesis to her.

Nijmegen, January 1997

Paul Frederiks

Chapter 1

Introduction

If you start me up, if you start me up I'll never stop
You can start me up, you can start me up I'll never stop

From *"Start Me Up"*,
The Rolling Stones

Ever since it reached a professional level, the field of software system development has been suffering from the so-called *software crisis* During the sixties attempts were made to solve this crisis with problem-oriented programming languages, like ALGOL 68 ([WMP+76]) and PASCAL ([Wir71]) These languages provide abstraction mechanisms both on the algorithmic and data level, but lack abstraction mechanisms for modular programming Languages with adequate facilities for "programming in the large", like SIMULA 67 ([DN66]), did not become popular, apart from ADA ([US83])

In the eighties a number of fourth generation programming languages like SQL ([NF84]) were introduced In combination with these languages, techniques were introduced for *conceptual modeling* Examples are ER ([Che76]) and its many variants, functional modeling techniques, such as FDM ([Shi81]), and so-called object-role modeling techniques, such as NIAM ([NH89], [Hal95]) Complex application domains, such as meta-modeling, hypermedia, and CAD/CAM, have led to the introduction of advanced modeling concepts, such as present in the various forms of Extended ER (see e g [TYF86], [EGH+92]), IFO ([AH87]), and object-role modeling extensions such as FORM ([HO92]) and PSM ([HW93], [HPW93]) Tremendous efforts were put in the introduction of development methods which cover the complete development life cycle, e g SDM ([TLH+88])

In the nineties a phenomenon re-arose[1] the *object-oriented* approach, OO for short An important argument that can be heard in favor of object-orientation is that its basic philosophy suits human problem solving techniques better than conventional programming languages and methodologies In a short time many[2] new object-oriented techniques and methods were

[1] Firmly rooted in SIMULA 67 and the software research of the seventies
[2] [Web94] contains already a list of over two hundred and forty references!

developed, like OOA ([CY90]), OMT ([RBP+91]), Booch ([Boo91]) and OOSE ([JCJO92]).

This plethora of techniques reflects the general situation in the field of information systems development. In [AF88] this situation is described by the term *Methodology Jungle*. In [Bub86] it is estimated that during the past years, hundreds if not thousands of information system development methods have been introduced. Most organizations and research groups have defined their own methods. Hardly any of them has a formal syntax, let alone a formal semantics. The discussion of numerous examples, mostly with the use of pictures, is a popular style for the 'definition' of new concepts and their behavior. This has led to *vague* and *unnatural* concepts in information systems development methods.

1.1 Focus of this thesis

The focus of this thesis is on the *analysis phase* in the context of the development of *information systems*.

From many articles in the literature and experiences in the field of information system development, it is well-known that the *early* phases (analysis phases) often turn out to be the bottleneck of systems development, since the acquisition of requirements is notoriously difficult (see e.g. [Dav90]). It is also a well-known fact that the later in the development process an error is detected, the more expensive it is to correct it (see e.g. [Dav90]).

In our point of view, the success of an information system depends on two important aspects and thus should be considered in the analysis phase of information system development:

1. *flexibility* of the information system, and
2. the way in which a user can *communicate* with the information system.

Since applications tend to become more and more complex, and always to be changing as a result of their ever changing environment, the need for *flexible information systems* is increasing (as pointed out in figure 1.1 sixty percent of the changes are changes in wishes of domain experts and changes in data format). Furthermore, the need for *communication-oriented information systems* is increasing. Users wants to communicate with the information system using their 'own' language, i.e. *(semi-)natural language*, and users want to apply graphical facilities to depict the result of their queries (see e.g. [PSB+94]).

Object-orientation seems to be a promising candidate for the development of flexible information systems, whereas using natural language, initializing and validating information system analysis, seems to be an interesting way to achieve communication-oriented systems.

In the sequel of this introduction the notion of object-orientation, and the idea behind natural language conceptual modeling are discussed in section 1.2 and 1.3, respectively, resulting in the problem statement of this thesis in section 1.4. Related work and the outline of this thesis are presented in sections 1.5 and 1.6, respectively.

1.2 The phenomenon of object-orientation

Object-orientation has become one of the major buzzwords in the field of computer science, but as stated in [Kin89] object-orientation is also interesting from a business point of view. As it is the case with many buzzwords and commercial slogans, object-orientation has been suffering from misconceptions, overestimation and glorification on one side, and at the same time from ignorance of its 'déjà vu' aspects at the opposite extreme. In [Dit90] a few (and as the author says, slightly exaggerated) symptoms are mentioned:

- *everybody cries for mature products that have it already (without always knowing what it exactly is and what it is potentially good for),*
- *in the marketplace (and sometimes in research too), everybody's system claims to have it, or at least everybody claims to work on it,*
- *old stuff reappears under the new label (which may, in some cases, even be justified!).*

Summarizing these three symptoms the following question has arisen:

What is object-orientation?

In the sequel of this section a short overview of the notion of object-orientation is presented, without having the pretension to give an all-embracing answer to the question above. This overview starts in section 1.2.1 with a discussion on the roots of object-orientation. Section 1.2.2 provides a summary of the main concepts and characterizations of object-orientation. The advantages and disadvantages of object-orientation are presented in section 1.2.3. A note on formal foundations and common terminology for object-orientation is described in section 1.2.4.

1.2.1 Descent of object-orientation

Prominent authors of books (e.g. [Boo91], [CY90] and [Gra94]) on the subject of object-orientation agree that the history of object-orientation really starts with the discrete event simulation language SIMULA 67 in Norway in 1967. This history continues with the development of the language Smalltalk in the 1970s. At that time there were no design methods, not to mention analysis methods, for object-orientation available. Object-orientation remained a research issue until the mid eighties, begin nineties. As stated in [Kos95a], the overwhelming popularity of the object-oriented development now is to a large extent caused by industry-wide disenchantment with the classical *waterfall model* of software development.

Boehm (cited in [Boo91]) mentions a few problems of the rigid application of the waterfall model. A first problem of the waterfall model is that it does not adequately address the concerns of developing program facilities and organizing software to accommodate change. The waterfall model also assumes a relatively uniform progression of elaboration steps. Furthermore, the waterfall model does not accommodate the sort of evolutionary development

made possible by rapid prototyping capabilities and fourth generation languages. Finally, Boehm notices that the waterfall model does not address the possible future modes of software development associated with automatic programming capabilities, program transformation capabilities, and 'knowledge-based software assistant' capabilities.

But the rise of object-orientation (design and analysis) has not only been caused by the disenchantment with the classical waterfall model and the existence of object-oriented programming languages. Object-oriented design and analysis has proven to be a unifying concept in computer science, applicable not only to programming languages, but also to the design and analysis of user interfaces, databases, knowledge bases, and even computer architectures ([Boo91]). Object-orientation reuses well-known and well-proven concepts, such as encapsulation, subtyping, inheritance, and polymorphism, from conventional programming and conceptual data modeling techniques. Object-oriented analysis and design thus represents an evolutionary development, not a revolutionary one; it does not break with advances from the past, but builds upon proven ones.

Remark 1.1

This remark provides a short impression of the origin of object-orientation from a programming language point of view ([Kos95b]).

Object-oriented languages have finally introduced practicing software engineers to the notion of abstract data type ([HK87]), which has for a long time been used by academics for the mathematical description of data types. The properties of an abstract data type are described together with its operations. The goal of an abstract data type is to be independent of any representation, and no assumption is made upon the structure of the data.

A module, as used in MODULA-2 ([Wir85]), can be seen as the extension of an abstract data type with a global state, in the form of a concrete data structure. Via import- and export-parameters it can be indicated what data structures and what operations may be used by other modules without any knowledge of how these data structures and operations are implemented. One might say that a module corresponds to an object type in an object-oriented approach, with the restriction that there is only one instance of a module available, which is globally defined.

A last remark on the differences between object types and modules is that an object type usually describes a smaller part of the application domain. A solution using object-oriented methods is therefore often of a finer granularity than a classical modular solution, which makes reuse more plausible.

1.2.2 Characteristics of object-orientation

Unfortunately, the authorities (on the subject of object-orientation) contradict one another wildly in formulating what terms and concepts make a modeling method or programming language object-oriented. Following [Gra94], [Boo91], [CY90], [Sny93], and [ADM+89] object-oriented software engineering can be characterized by the following terms and concepts:

1. *Object.* An object is a model of a concept in an application domain characterized by its behavior and internal state. Often used synonyms for objects are *instances* or *entities.* Objects with the same features, such as attributes (also referred to as properties) and operations (also referred to as methods), are grouped in so-called *object types* or *(object) classes.*

2. *Message.* A message represents communication between objects. This eliminates data duplication and ensures that changes to data structures encapsulated within objects do not propagate their effects to other parts of the system. Messages are often implemented as procedure calls.

3. *Encapsulation.* Data structures and implementation details of an object are hidden from other objects in the system. The only way to access the state of an object is to send a message that causes one of the methods to execute.

4. *Inheritance.* Objects inherit all and only the features of the classes they belong to, but it is also possible to allow classes to inherit features from more general superclasses. In this case inherited features can be overridden and extra features can be added to deal with exceptions. The notion of *multiple inheritance* refers to classes which inherit features from more than one superclass.

5. *Polymorphism.* The ability to use the same expression to denote different operations is referred to as polymorphism. Polymorphism is often implemented by *dynamic* or *late binding.* Inheritance is a special kind of polymorphism that characterizes object-oriented systems.

6. *Aggregation.* The notion of aggregation is used to express that an object is a composition of other objects. Such objects are also referred to as *complex objects* and are often represented as sets of other objects.

1.2.3 Pros and cons of object-orientation

This section summarizes advantages and disadvantages of object-orientation ([MO92], [CY90] and [Mey88]).

The following advantages of object-orientation are encountered:

- A better *integration* of data and processes. In many conventional analysis and design methods there is no satisfactory connection between the data models and process models. Object-oriented modeling techniques have natural integration of data and processes as a result of the encapsulation of attributes and operations in an object.

- Object-oriented systems are *loosely coupled.* A consequence of encapsulation of data and operations in objects is that objects can only access data of another object by sending messages. This results in an elimination of the dependence on global data. Global data is a mine of problems whenever several processes need access to this data.

- *Reuse* of classes. As a result of the encapsulation of data and processes (high cohesion), and the fact that object-oriented systems are loosely coupled, classes are stand alone-products. Classes can be put in *object libraries*, which stimulates the reuse of these classes[3].

- *Cheaper maintenance.* Encapsulation of data and processes makes it possible to change a class without changing other classes. Furthermore, before a class has to be changed the object library can be consulted for similar classes (generalization/specialization).

- *Better maintenance.* Conventionally developed information systems are implemented to fit as good as possible for the available hardware, whereas object-oriented implemented systems try to capture the reality as naturally as possible. This difference in point of view is very important if we realize that almost sixty percent of all changes of systems are changes in the wishes of the users and changes of the format of data whereas changes of hardware contribute only six percent to the changes of systems. See also figure 1.1 (adapted from [Mey88]).

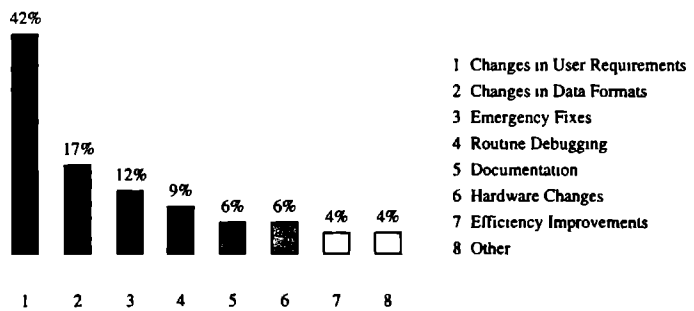


Figure 1.1: The costs of maintenance

- Better *communication* between system analyst and domain expert. Object-oriented methods try to describe the reality by objects which represent abstract and concrete notions of the problem area. Furthermore, in contrary with conventional methods, data and operations are grouped together in object-oriented models, which provides a better overview of the problem domain.

- Better *connection* between the stages of development. In contrast with conventional modeling techniques, models of object-oriented techniques for the different stages of the modeling process have better connections. The products of the different stages of conventional modeling techniques often need an extra translation for their interfacing.

[3]In [Hat96] the author argues, completely counter-intuitive, based on studies, that small components (classes) tend to have a disproportionately larger number of bugs than bigger components.

The following disadvantages of object-orientation are mentioned:

- *Informal foundation* of most object-oriented methods. As discussed in section 1.2.4 many conventional object-oriented modeling techniques lack a formal foundation. In general the syntax is explained by an enormous number of pictures, whereas the semantics is not defined at all.
- *New investments.* Before an organization can profit from the benefits of object-orientation, the organization has to be restructured for object-orientation. This includes training of personnel (analysts and management), acquisition of object-oriented Case tools and object-oriented software.
- Development for *future reuse.* In order to benefit from reuse in the future it can be necessary to develop classes which are not directly necessary for the current project. But too much insistence on providing for the future may lay down the project.
- Reuse of *bad products. If* a class contains mistakes, these mistakes can disorder all systems which use this class. Furthermore, each generalization or specialization of these 'bad' classes introduce more errors.

For more literature about the pros and cons of object-orientation the reader is referred to [PP94], where the authors describe quantitative and qualitative aspects of object-oriented development based on the experience gained in several projects.

1.2.4 Lack of foundation and common terminology

Many experts believe that the object-oriented approach is a step forward in solving the software crisis even though this approach lacks formal foundations and common terminology [4] ([ADM+89], [Dit90]). The problem is that everybody seems to have his own opinion about object-orientation[5]. For example, the definition of the notion object is already different for Coad and Yourdon, Booch, and Rumbaugh et al., see table 1.1. Hand in hand with a discussion on common terminology, consensus has to be made about which features (e.g. aggregation, polymorphism) object-oriented methods should have. For elaborate discussions on features of object-oriented methods the reader is referred to [EJW95] and [KM90].

From the literature a number of formal methods are known, such as TROLL ([JSHS96]), Object-Z ([SBC92]), and VDM++ ([DK92]). A major advantage of formalized methods is that models in these methods can be (possibly automatically) verified. However, formal foundations for object-oriented methods should not be a goal by themselves but a way to support the development process and to get a better understanding of the concepts of the method.

[4] As described in [Mey96], a consensus about common terminology seems to come closer since Booch, Jacobson, and Rumbaugh are developing the so-called *Unified Modeling Language* ([BR96a]) together.

[5] Fundamental discussions on the notion of object and object-oriented methods can be found in [Wig95] and [Wie91], respectively.

Coad and Yourdon	Booch	Rumbaugh et al.
An object is an abstraction of something in an application domain, reflecting the capabilities of a system to keep information about it, interact with it, or both; an encapsulation of attribute values and their exclusive services. (synonym: an instance)	*An object is something you can do things to. An object has state, behavior, and identity; the structure and behavior of similar objects are defined in their common class. The terms instance and object are interchangeable.*	*We define an object as a concept, abstraction, or thing with crisp boundaries and meaning for the problem at hand. Objects serve two purposes: They promote understanding of the real world and provide a practical basis for computer implementation. Decomposition of a problem into objects depends on judgment and the nature of the problem. There is no single correct representation.*

Table 1.1: A lack of common terminology

Many popular methods, such as Booch and Rumbaugh, can not strictly be considered formal methods because they only provide informal notations ([RDPS96]). These methods however, are easier in use by system analysts and better readable (and thus better to validate) by domain experts than formal methods like VDM++.

Remark 1.2

In the scientific and professional journals a number of interesting papers has appeared which discussed whether the industry benefits from the use of formalized methods.

[Fit96] presents the results of a in-depth field study which investigates the role of (formalized development) methodologies in practice. The paper discusses the reason why many practitioners do not use a formalized development methodology. The author observed that the degree of inertia on the development process is proportional to the degree of formality of the methodology.

The papers [Law96] and [Jac96] describe a discussion between the two authors on the subject of formalized methods. The first author, who is a software consultant, states that he likes informal methods, because these methods use natural language as a medium, which is familiar to both the system analyst and the domain expert. The author of the second paper, an assistant professor, states that natural language documents are poor depositories of the analyst's insights. He agrees that natural language is necessary for understanding the domain expert, but it is not sufficient to be able to record that understanding.

1.3 Natural language based conceptual modeling

Figure 1.2 shows a simplified view on the process of natural language based conceptual modeling. In order to initialize the modeling process the domain expert must provide the system analyst a natural language specification. The process of obtaining such an (initial) natural language specification is called *elicitation* or *requirements engineering*. The primary task of the system analyst is to map the sentences of the natural language specification onto concepts of a particular conceptual modeling technique. This is referred to in figure 1.2 by the arrow labeled *modeling*. The conceptual model in turn is translated to natural language sentences in order to be validated by the domain expert. This translation is called *paraphrasing*.

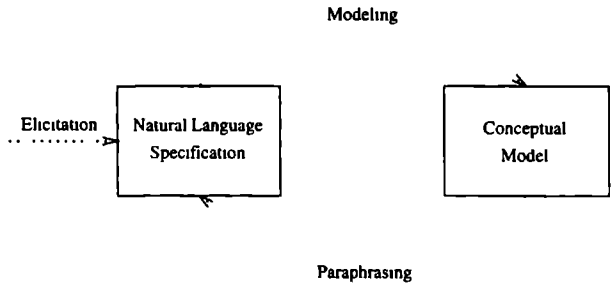


Figure 1.2: Natural language based conceptual modeling

This way of working is most effective when performed as an iterative incremental process, since it is impractical to elicit a complete and correct natural language specification all at once. Furthermore, since the domain expert is actively involved in the analysis process, such an iterative analysis process contributes to the user's confidence in the resulting information system, also referred to as *user satisfaction* (see e.g. [TY96]).

For many techniques based on using a natural language specification, the goal of the modeling process is to derive the grammar that governs the communication (*expert language*) within the so-called *Universe of Discourse*, or UoD for short ([Gri82]). This grammar, also referred to as *information grammar* (see e.g. [WZ96], [HPW94], [Lek93]), can be depicted as an information structure diagram. A paraphrase of the symbols of an information structure diagram forms the base for the terminal symbols for this grammar. A major advantage of the linguistic approach is that (intermediate) results can be readily validated by the informed user, as each intermediate result corresponds to a (partial) grammar for sentences in the language spoken in the UoD. As a result, the information grammar can be used as a basis for the query language of the user with the information system ([HPW94]).

Since the information grammar governs the communication within the UoD, it implicitly contains information about the relevant data and processes of the UoD. Therefore, the

information grammar can be used to identify the relevant data and processes. In traditional conceptual modeling techniques, such as ER, the information grammar only provides the structure of the data in the UoD. Query languages such as SQL are often used as a communication language of the user with information systems based on such information grammars (see e.g. [HE92]). Experience has shown that users have difficulties formulating SQL statements. There are two main reasons:

1. The syntax rules for SQL are too complex, especially because of their recursive nature.
2. The relation between a relational model and the corresponding UoD is usually not clear to users.

The first problem may be solved by allowing a semi-natural language format for the query language. An example of such a language is Lisa-D ([HPW93]). The second problem is addressed by conceptual modeling techniques such as NIAM, PSM, and FORM.

Remark 1.3

Using natural language for problem specification is not new in the field of computer science. In the early seventies syntactically-oriented programming methods, like step-wise refinement, were introduced, see e.g. [Dij76] or [Mee78]. The step-wise refinement method paraphrases the problem in a top-down fashion, using simple control structures like `IF THEN ELSE FI`. *Also the use of natural language in information modeling is not new. The conceptual data modeling techniques EER (as described in [BCD+95]), NIAM (as described in [DO90]), and PSM (as described in [HPW94] and [CW93]) are based on such an approach.*

Remark 1.4

[SS96] contains a number of interesting remarks about requirements engineering in general, and completeness of requirements in particular.

In their opinion, and we agree on this point, requirements engineering should encompass learning about the problem, understanding the needs of the potential users, discovering who the user really is (learning his/her language), and understanding all the constraints on the solution. This opinion is shared with Jackson ([Jac95]) who faults current software development methods for focusing on the characteristics and structure of the solution rather than the problem.

The incompleteness of requirements is acknowledged, at least by practitioners. Some may even claim that completeness in real-world requirements specifications is a utopian state about as achievable as getting it right the first time. Iterative conceptual modeling seems to be a good way to achieve as complete as possible a conceptual model of the UoD. As described in [BGM85] the use of conceptual modeling as a basis for requirements was a major signpost in directing researchers to this perspective. In [RP96] it is stated that incremental, iterative styles of system development form a major trend in system development methods (e.g. RAD methods).

1.4 Problem statement

Having discussed the nature and characteristics of object-orientation, as well as having discussed the idea behind natural language based conceptual modeling, we now formulate the subject of this thesis.

> *This thesis introduces a formal framework for the (1) derivation, (2) verification, and (3) validation of object-oriented analysis models, based on natural language, intended for the specification of flexible and communication-oriented information systems.*

1.5 Related work

Linguistic approaches to object-oriented modeling have been taken before. For the OOA method ([CY90]) a simple linguistic approach is described, which only considers *nouns* and *verbs*. Although this method offers the system analyst not more than a rule of thumb, it is still very useful for the determination of the global structure of the system, i.e. this method supports the analyst in finding object types and action types.

In the SACIS method (see e.g. [Gra94]) verbs are refined to: *doing verbs* (expressing an action type), *being verbs* (expressing a classification), *having verbs* (expressing composition), *modal verbs* (expressing conditions) and *stative verbs* (expressing an invariance-condition). Nouns are categorized into *proper nouns* which stand for instances and *improper nouns* which can be used for the determination of object types. Besides these two refinements the SACIS method provides grammatical views for attributes, operations, associations and events.

The KISS method ([Kri94]) is a further refinement, even though some parts of speech which are considered in the SACIS method are left out, because it allows for a deep analysis of the informal specification. Besides finding object types, via direct objects and indirect objects in sentences, the *connection* and *direction* between object types is investigated by focusing on verbs (predicates) and prepositions. *Adjectives* and *adverbs* are used for recognizing properties (attributes) of object types and action types. A novelty is the use of *gerunds*. (A gerund is a substantive noun that is derived from the infinitive form of a verb by suffixing this initial with '-ing'.) Gerunds turn out to be objectified action types.

All three methods use in some way grammatical analysis of the initial specification but lack a mechanism for paraphrasing the models obtained by these methods to natural language sentences. The way of working of these methods, during analysis, consists of collecting lists of candidate object types, action types, etc.

In [RP92] the *Chomsky theory* ([Cho65]) is used for the generation of natural language sentences out of conceptual schemata. The basic Chomsky assumption is the existence of a universal underlying structure to *any* sentence in *any* human language. In addition, there is an infinite number of ways, namely the surface structures, to represent the deep

structure in different languages The deep structure expresses semantics of a sentence by means of semantic elements and relationships among them The proposed framework is supported by an expert design system, known as OICSI (French acronym for intelligent tool for information system design) Note that this approach is not particularly focused on object-oriented analysis

A similar and according to the authors ([MG94]) less sophisticated tool is LOLITA, which supports a linguistic approach for developing object-oriented systems A data dictionary is used to find semantic relations between the object types that are detected in the informal specification In this way, for example, the object types *teacher* and *professor* can be related Furthermore, the alternative formulation *master* may be suggested by the system. Besides this feature, LOLITA is able to automatically identify ambiguities, inconsistencies, to correct misspellings and guess new words

In [Joh95] the author identifies related object types using Galois connections In order to verify that the proposed correspondences between object types are consistent with the semantics of the conceptual schemas, the modeling formalism utilizes linguistic instruments The instruments used are in particular *case grammars* and *speech act theory* The modeling formalism described can be used to support *schema integration*

The LIKE project (Linguistic Instruments in Knowledge Engineering) has resulted in a method called COLOR-X ([Bur96]) which is an abbreviation for COnceptual Linguistically based Object-oriented Representation language for Information and Communication Systems (ICS is abbreviated to X) The static and dynamic part of object-oriented modeling is described in [BR95b] and [BR95a], respectively The expert language is captured in graphical models which can be translated to an intermediate language called Conceptual Prototyping Language (CPL) CPL ([Dig89]) is a formal modeling technique, based on *functional grammars* ([Dik89]), which can be used for specification as close as possible to the informal specification As a result the paraphrasing of the graphical models is based on CPL

The main difference between the above mentioned paraphrasing approaches and ours is that they are semantically based whilst ours is syntactically based

1.6 Thesis outline

There are several ways of reading this thesis To get a quick impression one reads the introduction (chapter 1) and conclusions (chapter 8) In the precedence graph of figure 1 3 this is indicated by a solid arrow

The dashed arrows indicate a global way to read this thesis In chapter 2 the role of information grammars is discussed from an information system architecture point of view Furthermore, a terminological framework for information system development methods is presented which is of great importance for the succeeding chapters This chapter is mainly based on [FW96d] A way of working (building information grammars) and thinking, based on natural language and logbooks, is described in chapter 3 This chapter is a compilation

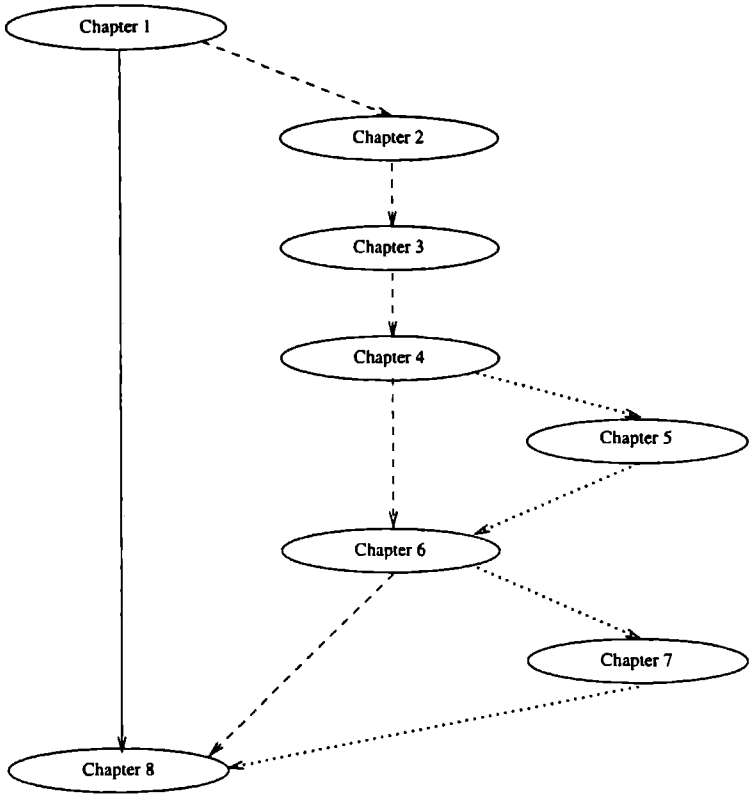


Figure 1.3: Different ways of reading this thesis

of [FW96b] and [FKW95]. The intuition and informal introduction of the object-oriented analysis models and their relation with the notion of logbooks are presented in chapter 4, Details of this chapter can be found in [BFW96], [FW96c], and [FKW96]. Finally, aspects with respect to validation, verification and design can be found in chapter 6. This chapter is based on [DFW96], [FW96c], and [FW96e].

The dotted arrows indicate a path for those readers who want to know more about the formalized parts of this thesis. The formalization and integration of the object-oriented analysis models presented in chapter 4 are discussed in chapter 5. Furthermore, the relation of the object-oriented analysis models and the underlying information grammar is presented. This chapter is based upon [FW96c]. Chapter 7 discusses a categorical framework for conceptual modeling techniques and was published in [HLF96] and [FHL97]. This framework is used to provide semantics to the object-oriented analysis models.

Chapter 2

Information Grammars

Yeah heard the diesel drummin', all down the line
Oh, heard the wires a-humming, all down the line

From: "All Down The Line",
The Rolling Stones

2.1 Introduction

In this chapter[1] we discuss: (1) the role of information grammars for the architecture of, the communication with, and the specification of information systems (section 2.2), and (2) a general terminological framework for information systems methods, and thus also for methods specifying information grammars (section 2.3). This framework is the basis of the information system development framework described in this thesis. Finally, in section 2.4, a short summary of this chapter is provided together with the reflection of how the subjects treated in this chapter are used in the succeeding chapters.

2.2 Information system architectures

The methods for information systems development have evolved in the course of time from machine-oriented into more human-oriented approaches. Rather than focusing only on the structure of the information, the way the information is used to capture the pragmatics of the communication language within the application domain has emerged into a major point of attention. This is reflected by a change of architecture, man-machine communication and analysis methods. Central for these aspects is the information grammar. This grammar is the basis for all communication with the information system but also describes the structure of all information to be processed.

In this section we discuss several typical architectures for information systems. Coupled to these architectures, we focus on man-machine communication. Before we discuss a general

[1]This chapter is partially based on [FW96d].

architecture of an object-oriented information system, the evolution of information system architectures is discussed shortly. This discussion is partially adopted from [Bub86] and is augmented with our own view on information system architectures.

2.2.1 File-oriented architecture

Figure 2.1 shows the view on an information system in the sixties. An information system consisted of a collection of application programs, i.e., data definitions and procedures (non-transparent), communicating with each other by the passing of files.

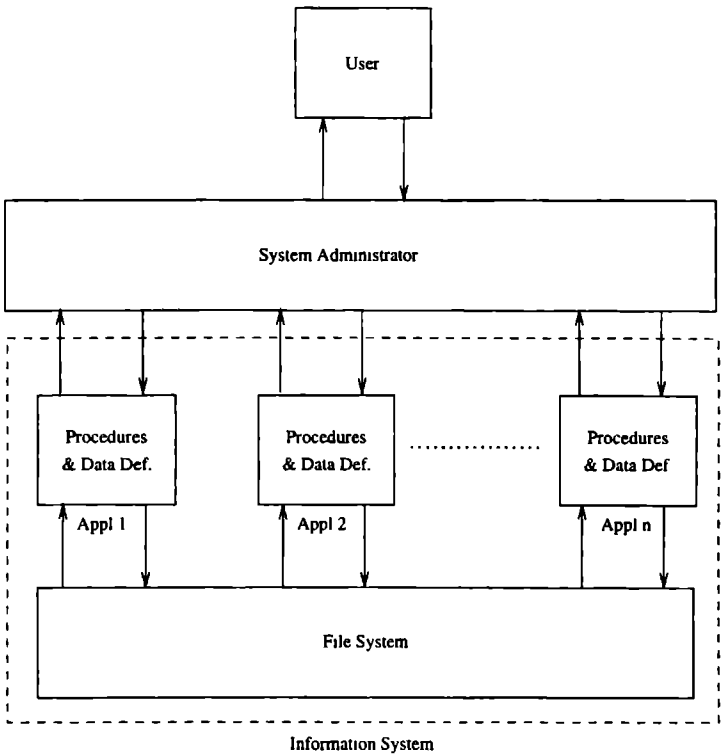


Figure 2.1: A file-oriented architecture of an information system

The operation mode of such an information system was mainly batch processing, usually dominated by a *system administrator*. The task of the system administrator can be summarized as: to administrate and guarantee the system at an operational level. The input data for a batch run were gathered in one or more files (sets of cards), the results consisted of files and printed output. The format of data files was rather straightforward. It was

the responsibility of the end user to transform real data into this strict format, and to interpret the resulting data within the application domain. The communication between man and machine thus was mainly on the side of the machine, the human had to adapt to the machine.

Building an information system concentrated around the construction of the programs in the collection. In other words, the programs had a central place, the data definition was of secondary importance ([Dat91]). For the construction of such systems process-oriented techniques for program construction were used (such as step-wise refinement).

2.2.2 Data-oriented architecture

In the seventies the prevailing view on information systems became data-oriented and was dominated by the information systems data definitions (see figure 2.2). The essential difference with the file-oriented architecture is the separation of data definition (in data bases) and data processing (in application programs). This separation was achieved by explicit data definitions in data base schemata at several levels: *external*, *conceptual* and *internal* ([Dat91], [Gri82]). Another difference with the file-oriented architecture is the absence of a system administrator. However, in the data-oriented architecture a *database administrator* was responsible for the integrity and structural data aspects on all levels. The database administrator also governed authorization aspects via the external user views.

Note that the data-oriented architecture is not the same as presented in [Dat91]. In our architecture the aspects of user communication are also involved, while details at the level of data storage are omitted.

The communication in the data-oriented architecture is typically guided by filling a (digitized) *form*. Forms are a very widespread communication mechanism within organizations and therefore very acceptable for human beings. In [PSB+94] it is argued that form filling is suitable from an ergonomic point of view. In practice, forms are usually very well designed, and close to the *mental model* of the (intended) users ([Sch92]).

In file-oriented architectures, it was the responsibility of the user to translate the data from (physical) forms into the format described by the data files. In the data-oriented architecture, physical forms were mapped onto digitized forms, enabling the user to simply fill a form in known frame of reference. The process of data extraction from the form was now performed by the computer rather than the user. As a result, the man-machine communication became much more directed towards the human being.

As a result of this approach, a separation of concerns is possible. Basically, an application program is divided into the following parts:

- the interface with the user.
- the interface with the data base.
- the processing in terms of these interfaces.

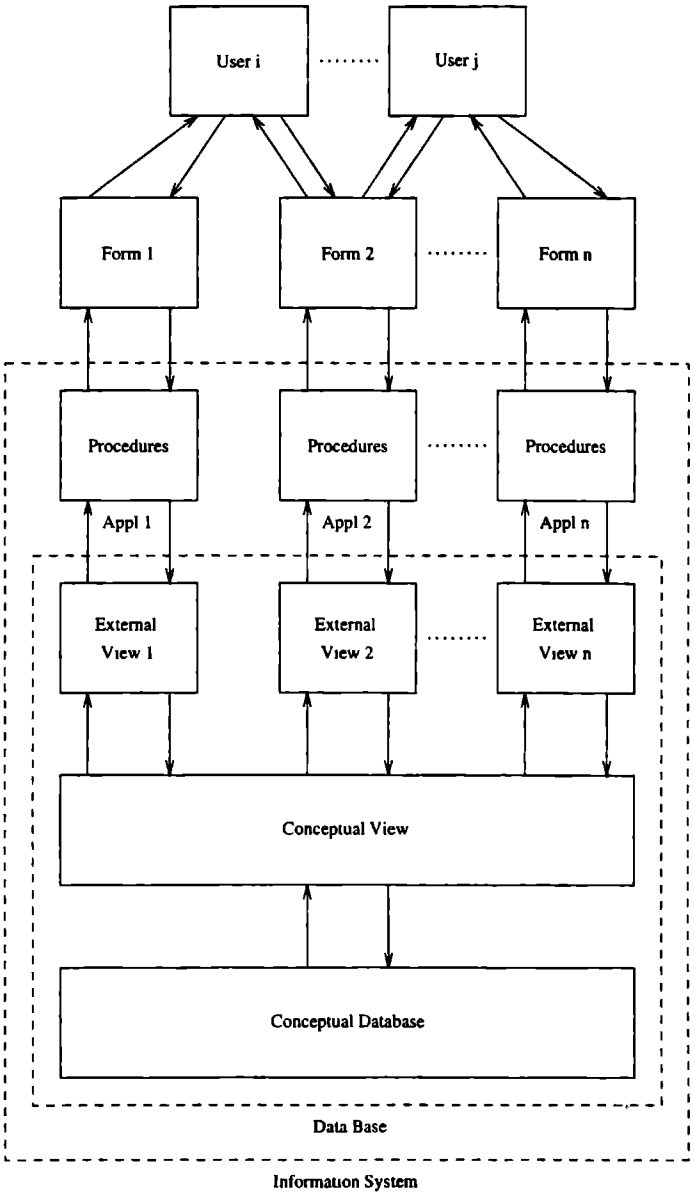


Figure 2.2: A data-oriented architecture of an information system

Central is the construction of the data models, which are not dependent of any application program. Several methods and techniques have been designed to support this construction process, e.g. the relational model ([Cod70]) and ER ([Che76]). Special utilities are usually available to build forms. However, most of them do not support the designer at the level of making well-designed forms. A weakness in all data-oriented analysis methods is their inability to describe the process side adequately.

2.2.3 Communication-oriented architecture

During the eighties the data-oriented architecture was still widely used. However, this view was augmented with all kinds of fourth-generation language mechanisms (query language facilities, form definitions and processing definitions, report generators, dictionaries, all kinds of graphics and spread-sheets, etc.). The role of the information system changed from a dedicated system into a general purpose system. The system now keeps track of the definition of the data, and also administers the application programs (the *procedures base*) and the actual forms (the *forms base*).

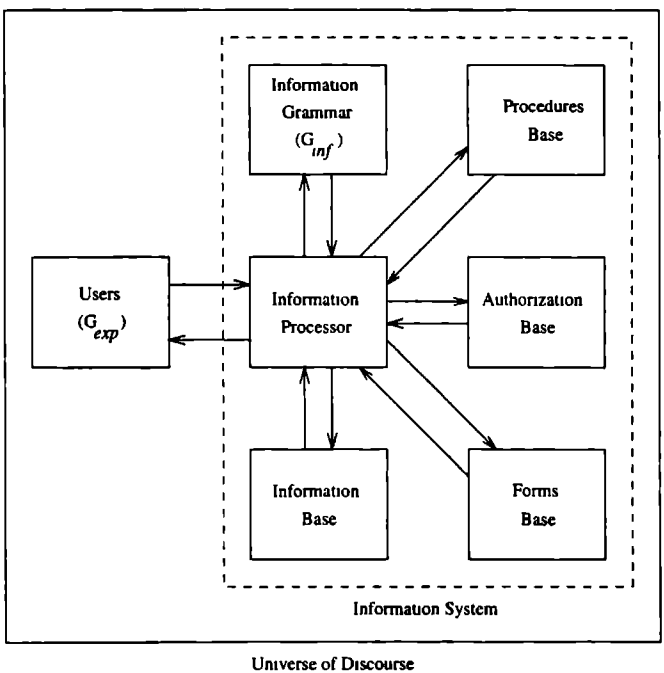


Figure 2.3: A communication-oriented architecture of an information system

A general communication-oriented architecture for information systems is depicted in fig-

ure 2.3 (which is an extended version of the architecture presented in [Win90]). In this architecture the user sends update or retrieval requests, in accordance with a certain *information grammar* [2], to the *information processor*. Each request is checked against the user's access rights from the *authorization base*. If a user is authorized to perform a request, the information processor performs the request on the *information base*. In [BW94] the topic of authorization is discussed extensively.

The communication-oriented architecture offers the user much more freedom for manipulation and interpretation of data. Due to the extra facilities, the system now is capable of providing the user with interpreted data, for example in the form of a pie chart. Therefore the communication with the information system became the main topic for improvement. A typical communication-oriented system is an SQL system. The communication between human and machine is in terms of the elements that describe the application domain, according to the syntax rules of SQL (see e.g. [NF84]). In order to be able to communicate with the system, the user should know:

- the vocabulaire: the names of tables and their columns,
- the grammar rules: the syntax rules of SQL.

Both components are part of the information grammar.

Data modeling techniques such as NIAM ([NH89], [Hal95]), PSM ([HW93]), and FORM ([HO92]) focus on the communication (*expert language* [3]) within the application domain rather than on the required data definition. As they model the communication, (partial) results can be validated by paraphrasing their models to natural language sentences. Furthermore, many methods and techniques are supported by Case tools and Case shells ([HV96]).

2.2.4 Object-oriented architecture

In an object-oriented approach, the centralistic view of the communication-oriented architecture is completely modified (see figure 2.4). The information system provides the user with the opportunity to address objects more or less directly. The information system consists of subsystems (complex objects) which communicate by message. Each such communication is governed by a corresponding local communication grammar. The local communication grammars together form the (global) information grammar.

In figure 2.4 subsystem i sends messages to subsystem j according to grammar $\mathcal{G}_{ij}$. Subsystem j responds with messages to subsystem i according to grammar $\mathcal{G}_{ji}$. A subsystem in its turn may consist of a number of other subsystem. The smallest subsystem is an (elementary, i.e. non-composed) object. The information system is a (possibly infinite) network of objects.

[2] Also abbreviated as G_{inf}.

[3] The expert language is captured by the so-called *expert grammar* which is denoted as G_{exp}.

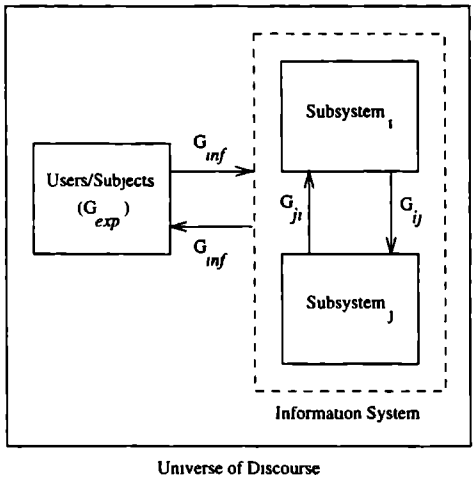


Figure 2.4: An object-oriented architecture of an information system

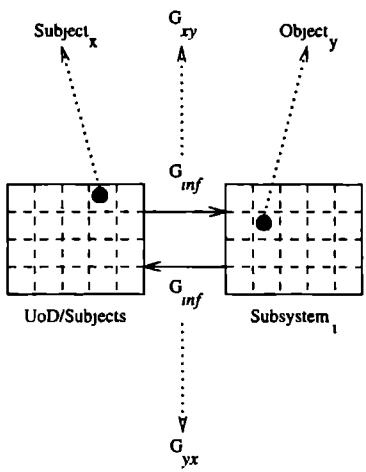


Figure 2.5: Subject/object communication

If we take a closer look at the subject/object communication, we see that the user (a so-called *subject*, i.e. an object outside the information system) makes update and retrieval requests by sending message to an appropriate object (see figure 2.5). Just like the communication between subsystems, the communication between subject x and object y is governed by a pair of grammars $\langle \mathcal{G}_{xy}, \mathcal{G}_{yx} \rangle$ which is suitable for the subject and the object respectively. An authorization mechanism is therefore not explicitly mentioned as the authorization is dominated by (and encapsulated in) the subject/object grammars.

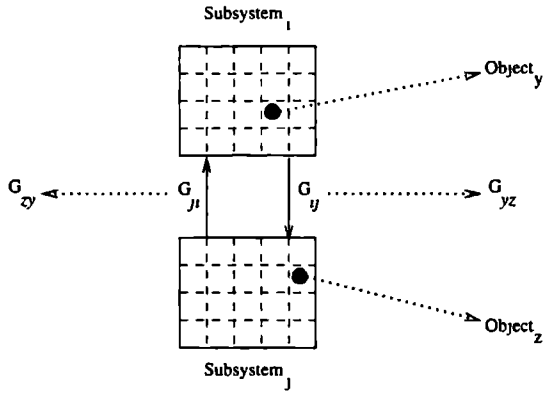


Figure 2.6: Object/object communication

A subject activates objects to perform actions, or creates new objects for this purpose in correspondence with its communication grammar. Activated objects y may, in turn, send new requests in accordance with some object/object communication grammar $\mathcal{G}_{yz}$ to other objects z (see figure 2.6). The result of such a request to another object is interpreted by the object, which was activated by the subject, and communicated to the user. The strict separation between data and procedures is dropped in an object-oriented architecture of an information system and is replaced by a strict encapsulation of data and operations related to an object type.

In the object-oriented architecture, the communication between human and machine becomes a communication between subjects and objects. An advantage over the communication-oriented approach is that the object-oriented approach provides the opportunity to differentiate more between cases of communication. For example, a lawyer may address the information system quite differently from a car seller. There is no a-priori need that they know each others language, but they may still want to communicate with the same information system.

The techniques of the communication-oriented architecture are still valid in this case. However, this leads to a number of local grammars that have to be integrated to form the global information grammar. For such integration there seems to be no good solution at the moment. The object-oriented modeling technique described in chapters 4 and 5 focus on the integration of the local grammars.

2.3 Terminological framework for methods

In this section a framework is presented which provides a better understanding of information systems development. This framework structures those aspects that can be distinguished within an information system development method. In this thesis the framework

is used to position the research needed in the development of communication, and object-oriented information systems.

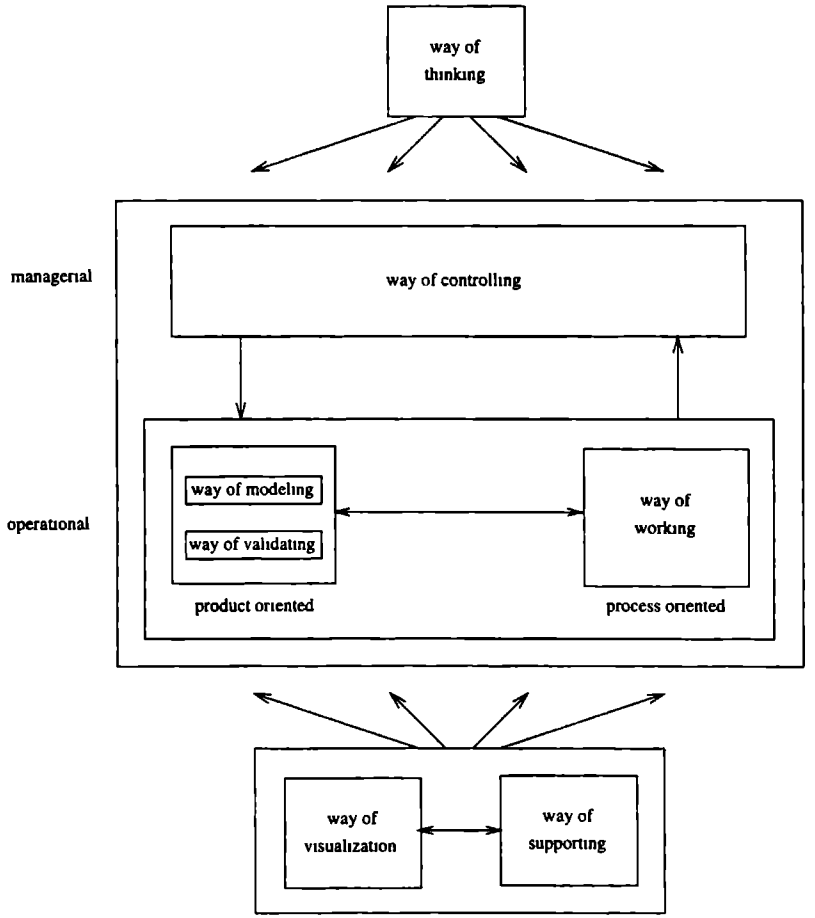


Figure 2.7: Terminological framework

The framework is based on Wijers framework ([Wij91]) and extended with the notion of *the way of validating*, and *the way of visualization* which is adopted from [Pro94]. Figure 2.7 presents the extended framework graphically. The arrows in figure 2.7 mean intuitively that the source of an arrow supports its destination. Note that the way of modeling and visualization together are usually referred to as a *modeling technique*. This framework consists of:

1. The *way of thinking* verbalizes the essentials, assumptions, and viewpoints of the method with respect to the problem domains, solutions and modelers. This notion can be compared with what has been referred to as *Weltanschauung* ([Sol83]), or *underlying perspective* ([Mat81]), or *philosophy* ([AF88]).

2. The *way of controlling* refers to the managerial aspects of the method. This includes the assignment of tasks and subtasks to manpower, quality insurance and process visibility, i.e. overall project management (see also [Sol88] and [Ken84]).

3. The *way of working* indicates the way in which the information system is developed, i.e. the overall strategy for the development of the information system. This strategy deals with the identification of tasks and subtasks in the development process and their feasible order.

4. The *way of modeling* describes the modeling concepts used in the method, the way these concepts are related (for verification, i.e. *are we building the model right?*), and the properties of these concepts. It structures the models which can be used in the information system development, i.e. it provides an abstract language in which to express the models.

5. The *way of validating* encompasses a way to validate (*are we building the right model?*) the models of the used techniques of the method, i.e. do the models meet the intended requirements of the problem domain. In contrast with verification, this cannot be checked automatically, as the intended requirements are only available in an informal format. However, validation can be supported with tools for *Rapid Application Development* (*RAD*), see e.g. [TCF90], and natural language paraphrasing (see e.g. [Dal92] and chapter 6).

6. The *way of supporting* of a method deals with all possible support for performing and facilitating systems development tasks, and can thus be defined as the collection of all possible tools. Tools are either automated (database management systems, code generators) or not automated (white-boards, paper and pencil). A generally accepted name for automated tools is *Case tool* (see e.g. [BW91] and [McC89]) or *Case shell* ([VH95] and [VHW91]).

7. The *way of visualization* describes how the notions of the way of modeling are visualized. Usually, models are represented in a graphical way (diamonds and squares in ER, and squares and circles in NIAM). It may very well be the case that different methods are based on the same way of modeling, and yet use different graphical notions (see also chapter 7).

2.4 Summary and outlook

The focus in this chapter has been on (1) the evolution of information system architectures and the role of information grammars in information system architectures, and (2) a

terminological framework for information system development methods.

The rest of this thesis is guided by this terminological framework. Chapter 3 discusses a way of thinking, working and controlling with respect to modeling information grammars. The intuition behind the way of modeling and visualization is mainly presented in chapter 4. Formal aspects with respect to the way of modeling are the topics of chapters 5 and 7. Finally, the way of validating and supporting is demonstrated in chapter 6.

Chapter 3

Building Information Grammars

You can't always get what you want
But if you try sometimes
You just might find
You get what you need

From: "You Can't Always Get What You Want",
The Rolling Stones

3.1 Introduction

In this chapter[1] some aspects of the terminological framework of section 2.3 are applied to the framework as presented in this thesis. The way of thinking and working is elaborated, and a short note on the way of controlling is presented.

In order to participate in a particular way of working the participants in the modeling process must possess particular skills which are closely allied to this way of working. The base axioms in section 3.2 focus on such skills required for natural language based conceptual modeling. Furthermore, these base axioms provide a justification for this way of working, showing that it maximizes the pros and minimizes the cons of using natural language as a specification language. We argue that the iterative process eventually leads to the intended conceptual model. The natural language based modeling process as sketched in section 1.3 is further refined in section 3.3 into a number of smaller steps. Each such a step will be discussed shortly. From the literature (see e.g. [CM96]) it is well-known that conceptual modeling methods are extremely useful for large scale projects. In general, such projects require a number of project managers to control and guard the progress of the modeling process. Section 3.4 contains a short note regarding managerial aspects of these methods. Finally, a summary of this chapter and an outlook to the succeeding chapters is presented in section 3.5.

[1] This chapter is based on parts of [FW96b] and [FKW95].

3.2 Cognitive requirements

Many conceptual modeling methods have as point of departure a description in natural language of the world to be modeled (UoD). This description is referred to as the *initial specification* and is provided by so-called *domain experts*. Ideally, this initial specification is a precise description from which *system analysts* can derive the required information system. However in practice this is in general not the case. As a consequence, a more elaborated modeling process is called for, which requires a number of skills from both domain experts and system analysts.

In section 3.2.2 these skills are investigated and presented as *base axioms* ([Nij89]). In order to be able to appreciate these base axioms the pros and cons of using natural language in a conceptual modeling process are presented in section 3.2.1. In section 3.2.3 it is shown that the base axioms help to overcome the drawbacks of natural language as a starting point for conceptual modeling. For more readings about using natural language for information system engineering the reader is referred to appendix A.

3.2.1 Pros and cons of natural language

Natural language has the potential to be a precise specification language *provided* it is used well. But there are not many people who can use natural language in a consistent, non-verbose, expressed on a uniform level of abstraction, complete, and unambiguous way.

Still, as stated in [Qui60], natural language is the vehicle of our thoughts and their communication. Since good communication between system analyst and domain expert is essential for obtaining the intended information system, the communication between these two partners should be in a common language: natural language. Initial specifications may also give hints on the way in which a user wants to communicate with the information system. A formal specification can never capture the *pragmatics* of a system. As a final argument, the formal specification may very well be paraphrased in natural language, which increases the possibility for domain experts to validate the formal specification (see chapter 6).

Remark 3.1

In [Dal92] more arguments for paraphrasing a conceptual model in natural language are given. Some of these arguments are more or less included in the arguments given above. These arguments are:

- *to lower the conceptual barrier of the domain expert,*
- *to ease the understanding of the conceptual modeling formalism for the domain expert,*
- *to ease the understanding of the UoD for the system analyst,*
- *to detect errors and traps in the conceptual model,*
- *to focus on certain aspects of a conceptual model, and*

- *to teach the conceptual modeling formalism to the domain expert or system analyst by paraphrasing the meta-model of the modeling formalism to natural language.*

3.2.2 Base axioms

The rationale behind natural language based conceptual modeling is a scheme of base axioms describing the cognitive identity of domain experts and system analysts. Roughly, a domain expert can be characterized as someone with (1) superior detail-knowledge and (2) minor capabilities for abstraction. The characterization of a system analyst is the direct opposite.

An example of a base axiom for a domain expert is the *completeness base axiom*:

Domain experts can provide a complete set of significant sample sentences.

As trivial as this axiom might seem, its impact is rather high: the axiom is the foundation for correctness of the information system, and provides insight into the requirements for those who communicate the problem domain to the system analyst. As the completeness base axiom is in practice too strong a requirement for a domain expert this axiom is weakened into the *provision base axiom*:

[D-1] *Domain experts can provide any number of significant sample sentences.*

As this base axiom does not state that a domain expert can provide a *complete* set of significant examples by a single request from the system analyst, some more base axioms that describe aspects of the communication between domain experts and system analysts are necessary.

A prerequisite for conceptual modeling is that sentences are elementary, i.e. not divisible without loss of information. As a system analyst is not assumed to be familiar with the semantics of sample sentences, it is up to the domain expert to judge about splitting sentences. This is expressed by the *splitting base axiom*:

[D-2] *Domain experts can split sample sentences into elementary sentences.*

A major advantage *and also* drawback of natural language is that there are usually several ways to express one particular event. For example, passive sentences can be reformulated into active sentences. By reformulating all sample sentences in a uniform way a system analyst can detect important syntactical categories during grammatical analysis of these reformulated sample sentences. The domain expert is responsible for reformulating the sample sentences, which is express by the *normalization base axiom*:

[D-3] *Domain experts can reformulate sample sentences into a unifying format.*

In order to capture the dynamics of the UoD the sentences need to be ordered. As a result, the domain expert has to be able to order the sample sentences. This is captured in the *ordering base axiom*:

[D-4] *Domain experts can order the sample sentences according to the dynamics of the application domain.*

During the modeling process, the information grammar (and thus the conceptual model) is constructed in a number of steps. After each step a provisional information grammar is obtained, which can be used in the subsequent steps to communicate with domain experts. First a description of the model so far can be presented to the domain experts for validation. In the second place, the system analyst may confront the domain expert with a sample state of the UoD for validation. The goal of the system analyst might be to detect a specific constraint or to explore a subtype hierarchy. This is based on the *validation base axiom*:

[D-5] *Domain experts can validate a description of their application domain.*

During the analysis process, the system analyst may check completeness by suggesting new sample sentences to the domain expert. This is based on the *significance base axiom*:

[D-6] *Domain experts can judge the significance of a sample sentence.*

Besides studying the cognitive identity of domain experts it is necessary to investigate the cognitive identity of system analysts. A first axiom which is applicable for system analysts is the so-called *consistency base axiom*:

[A-1] *System analysts can validate a set of sample sentences for consistency.*

A major step in a natural language based modeling process is the detection of certain import syntactical categories. Therefore a system analyst must be able to perform a (possibly partially automated) grammatical analysis of the sample sentences. The result of this grammatical analysis is a number of related syntactical categories. This leads to the *grammar base axiom*:

[A-2] *System analysts can perform a grammatical analysis on a set of sample sentences.*

A system analyst is expected to make abstractions from detailed information provided by domain experts. By having more instances of some sort of sentence, the system analyst will get an impression of the underlying *sentence structure* and its appearances. The sentence structures all together form the structure of the information grammar. This is addressed in the *abstraction base axiom*:

[A-3] *System analysts can abstract sentence structures from a set of related syntactical categories.*

As abstraction of sentences is based on the existence of a number of concrete sample sentences the system analyst must be able to generate new sample sentences which are validated by the domain experts, i.e. the *generation base axiom*:

[A-4] *System analysts can generate new sample sentences.*

Finally, a system analyst must be able to match sentence structures found with the abstraction base axiom with the concepts of a particular conceptual modeling technique. This is expressed by the *modeling base axiom*:

[A-5] *System analysts can match sentence structures with concepts of a modeling technique.*

Remark 3.2

The axioms presented for the system analysts focus on a natural language based modeling process. Of course, system analysts and system designers also need expertise for using and understanding modeling techniques. In [STR95] a conceptual framework is proposed for examining these types of expertise. The components of this framework are applied to each phase of the development process, and used to provide guidelines for the level of expertise developers might strive to obtain.

3.2.3 Impact of base axioms on natural language usage

In section 3.2.1, a number of disadvantages of using natural language for conceptual modeling has been encountered. Summarizing, natural language is hard to use:

1. in a *complete* way,
2. in a *non-verbose* way,
3. in an *unambiguous* way,
4. in a *consistent* way, and
5. on a uniform level of *abstraction.*

In this section we make it plausible that the base axioms for domain experts and systems analysts, introduced in section 3.2.2, can reduce the impact of the above mentioned disadvantages of using natural language.

Base axiom D-1 overcomes the problem with respect to the completeness of natural language specifications. The axiom states that domain experts can provide any number of

significant sample sentences. Assuming that each UoD can be described by a finite number of structurally different sample sentences, the probability of missing some sentence structure will decrease with each sample sentence generated. The process of providing sample sentences by domain experts is triggered, controlled and guided by the system analyst, as stated in base axiom A-4, to obtain convergence.

Specifications in natural language tend to be verbose. Complex (verbose) sentences will be offered to the domain expert for splitting (base axiom D-2) and their relevance for the problem domain (base axiom D-6). A natural language specification may also be verbose due to a large number of sample sentences. This is solved by the skill of system analysts to abstract from superfluous sample sentences (see base axiom A-3).

An often raised problem of natural language is ambiguity, i.e. sentences with the same sentence structure having a different meaning. In order to detect ambiguities, the system analyst may offer the domain expert alternative formulations of sentences for validation (base axioms A-4 and D-5). On the other hand, a system analyst may also elicit further explanation from a domain expert by asking to provide alternative formulations or more sample sentences with respect to the original ambiguous sample sentence (base axiom D-1).

In base axiom A-1 it is stated that a system analyst is equipped with the ability to verify a natural language specification for consistency. Just like the entire conceptual modeling process, consistency checking of natural language specifications has an iterative character. Furthermore, consistency checking requires interaction with the domain expert, as a system analyst may have either a request for more sample sentences (base axiom D-1), or a request to validate new sample sentences (base axioms A-4, D-5 and D-6).

Sentences of a natural language specification are often expressing several levels of abstraction. As a system analyst has limited detail knowledge, and thus also limited knowledge at the instance level, a prerequisite for abstraction is typing of instances (base axioms A-2 and A-3). As an example of such a sentence consider:

The Rolling Stones record the song Paint It Black.

The analysis made on such a sentence by the system analyst is in fact a form of *typing*, attributing types to each of its components. Some instances will be typed by the domain expert (the song *Paint It Black*) while others are untyped (*The Rolling Stones*). This may be resolved by applying a *type inference mechanism* to untyped instances (see section 4.2.5). Typed sentences can be presented to the domain expert for validation (base axiom D-5).

3.3 Modeling process

In this section the simplified view on the modeling process as depicted in figure 1.2 is refined. The philosophy behind the way of working is to use natural language as much as practicably possible in order to obtain the information grammar. Traditionally, three aspects of (natural or artificial) language are distinguished: *syntax*, *semantics* and *pragmatics*. Our approach

is to exploit syntactical aspects as much as possible, although some semantics may be necessary for the modeling process. The domain expert is held responsible for the semantic part. Other approaches such as COLOR-X ([Bur96]) put the central emphasis on semantics.

3.3.1 Way of working

The way of working which is guided by the skills (section 3.2.2) of domain experts and system analysts, can be further refined into the following stages:

1. Collect significant information objects from the application domain.
2. Verbalize these information objects in a common language. This step results in an *initial specification* and is guided by base axiom D-1.
3. Reformulate the initial specification into a unifying format. This step results in an *informal specification* and is guided by base axioms D-2, D-3 and D-4.
4. Discover significant syntactical categories and their relationships. This step results in a *normal form specification* and is guided by base axiom A-2.
5. Abstract sentence structures from the normal form specification and match these sentences structures to concepts of a modeling technique. This step results in *analysis models* and a sample *population* and is guided by base axioms A-3 and A-5.
6. Interpret the analysis models and sample population. This step results in an *information grammar* and *lexicon* and is guided by base axiom A-5.
7. Produce by paraphrases a textual description of the conceptual model using the information grammar and the lexicon. This step results in a *textual description* and is guided by base axiom A-4.
8. Validate the textual description by comparing this description with the informal specification. This step is guided by base axioms A-1 and D-5.

These stages are repeated until the textual description of the conceptual model captures the informal specification. Figure 3.1 resumes the modeling process schematically. Each stage of the modeling process is marked with the number associated above and is discussed in the sequel of this section.

3.3.2 Verbalizing information objects

The communication in the UoD may employ all kinds of information objects, for example text, graphics, etc. However, a textual description serves as a unifying format for all different media. Therefore the so-called *principle of universal linguistic expressibility* ([Adr92]) is a presupposition for this modeling process:

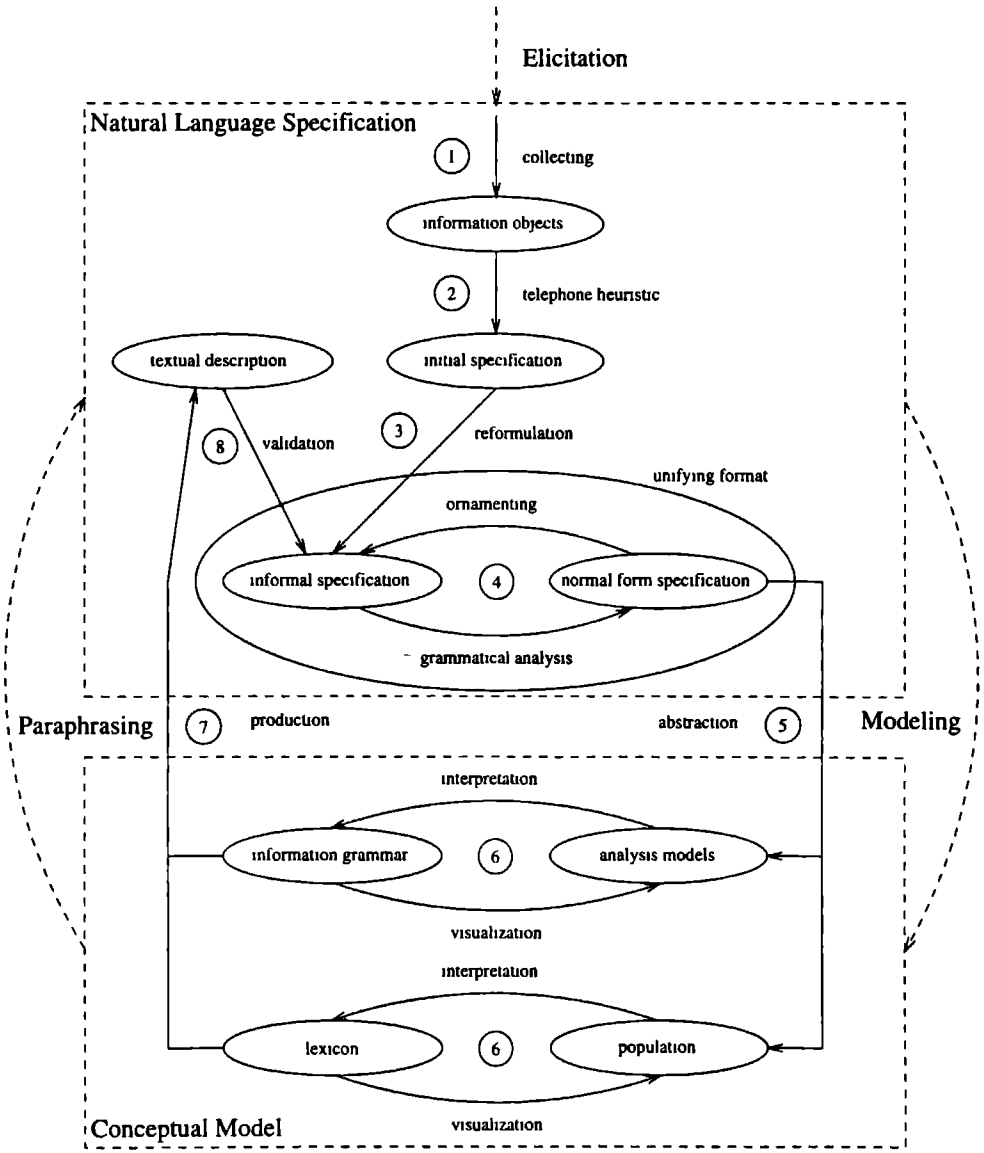


Figure 3.1: Overview of the analysis phase

All relevant information can and must be made explicit in a verbal way.

This principle may be concretized by the *telephone heuristic* ([NH89]):

Explain your observations to a non-expert via a telephone.

The principle of universal linguistic expressibility excludes common sensorial experiences, i.e. common sense reality. The domain expert and system analyst have no other experience in common than that which can be verbally expressed. The telephone heuristic provides the domain expert a way to verbalize the events occuring in the UoD in order to obtain an initial specification. Table 3.1 shows an example of an initial specification.

Mick Jagger and Keith Richards set up band 'The Rolling Stones'.
Mick Jagger and Keith Richards write Paint It Black.
Paint It Black is recorded, on tape number 666 in studio number 11, by The Rolling Stones.
The recording of Paint It Black, on tape number 666 in studio number 11, is produced by Mick Jagger and Keith Richards.
The song 'I Want You' is written by A. Knijff.
P. Frederiks and A. Knijff set up The Playful Plebs.
The band 'Playful Plebs' record, in studio number 2 on tape number 3, the song 'I Want You' and the song 'Long Way To Go'.
H. Honer produces the recording, recorded in studio number 2 on tape number 3, of the song 'I Want You'.

Table 3.1: Example initial specification

The language which results after the telephone heuristic is called the *expert language.* The underlying grammar is referred to as G_{exp} (see also section 2.2.3). The information modeling problem then can be described as:

Find, within a certain class of sufficiently efficiently computable grammars, an information grammar G_{inf} *which best approximates* G_{exp}.

Note that this problem ("grammar inference") is known to be NP-complete in general, but under certain restrictions it may become tractable ([Adr92]).

3.3.3 Unifying format for initial specifications

Only few conceptual modeling methods provide the system analyst with clues and rules which can be applied to the initial specification, such that this specification can actually

be used in the modeling process. Examples of these modeling methods are NIAM ([Hal95], [NH89]) for conceptual data modeling, and KISS ([Kri94]) for object-oriented modeling.

In order to structure this way of working, the (sentences in the) initial specification should be in accordance with some unifying format (referred to as the *informal specification*). In NIAM these sentences are called *elementary sentences*, while KISS uses the term *structure sentences*. Elementary and structure sentences provide a simple and effective handle for obtaining the underlying conceptual model of so-called *Snapshot Information Systems* (see e.g. [Dat91]), i.e. information systems where only the current state of the UoD is relevant.

However, even though these informal specifications are an important aid in modeling information systems, they are still too poorly structured. This lack of structure results in several problems during the modeling process. As an example, a system analyst has to come up with a lot of properties of the UoD which do not follow directly from the informal specification. One of these properties is the order and history of events that occurred. More concretely, since the mutual order of the sentences in an informal specification is lost, the analyst has to reconstruct this order. Other UoD properties are for instance related to the associates involved in events, and the role in which they are involved.

The notion of *logbook* is introduced as a common basis for various models to be produced during system analysis. A logbook has a unifying format which contains a complete description of the history of some UoD. Generally a logbook will report on changes of *instances* within the UoD, as well as on changes of the *structure* (information structure) of the UoD. A logbook may thus be seen as the sequence of all events which describe the evolution of the UoD. From a logbook each past state of the UoD can be derived, including the current state. As a consequence, a UoD basically corresponds to the set of all acceptable logbooks. The goal of the modeling process then is to obtain a formal description of this set of acceptable logbooks.

The logbook is intended as a structuring mechanism for informal specifications in the context of Snapshot Information Systems as well as *Temporal Information Systems* ([Sno90]). Furthermore, it supports the development of *Evolving Information Systems* (e.g. [PW95a], [Tre91], [NR89]).

In the sequel of this section structured informal specifications are discussed. Furthermore, a way to analyze these structured informal specifications is sketched. The introduction of logbooks is postponed until chapter 4.

Remark 3.3

The history of a UoD may also be described by an application model history, such as introduced in [PW95a]. In this view, the evolution of a UoD is seen as the evolution of its elements. An element evolution describes the state of the element at each point of time. This dual vision on evolving information systems will not be elaborated further in this thesis.

3.3.3.1 Structured informal specifications

A main goal of object-oriented analysis is to identify the objects, their relations, and their *responsibilities*, i.e. *which* object type is responsible for *what* action type. Furthermore, the execution order of the action types is relevant in order to describe the behavior of the information system, whereas properties of object types describe the state of the information system.

As discussed in section 1.5 the sentences of an initial specification provide many clues in finding object types, responsibilities, etc. In the KISS method the object type responsible for an action type is found by activating the sentences of an initial specification. In such an activated sentence the verb corresponds with the action performed, the subject with the object responsible for the action, and the nouns refer to candidate objects. For example, the sentence:

> *The song 'I Want You' is written by A. Knijff.*

is a passive sentence. This sentence can be rewritten into the following active sentence:

> *A. Knijff writes the song 'I Want You'.*

For the modeling process it is important to have sentences in an active formulation. However, the structure of passive sentences can also be included in the information grammar in order to obtain a more elaborated information grammar, and to obtain a better readable textual description of the information grammar (chapters 5 and 6).

Properties of object types can occur in natural language sentences as adjectives, adverbs, proper nouns, and in subordinate clauses. In order to obtain a most general unifying format for an informal specification, nouns in subordinate clauses seem to be a good candidate for expressing properties of objects and actions.

The sentences of the initial specification need to be elementary. For example the sentence:

> *The band 'Playful Plebs' record, in studio number 2 on tape number 3, the song 'I Want You' and the song 'Long Way To Go'.*

can be split into the sentences:

> *The band 'Playful Plebs' record, in studio number 2 on tape number 3, the song 'I Want You'.*

> *The band 'Playful Plebs' record, in studio number 2 on tape number 3, the song 'Long Way To Go'.*

The sentence:

Mick Jagger and Keith Richards set up band 'The Rolling Stones'.

can not be split since *Mick Jagger* and *Keith Richards* have set up *The Rolling Stones* together.

As stated in base axiom D-4 the order of the sample sentences provides clues for the dynamics of the UoD. Whereas the other reformulations focus on one sentence, this step requires insight in the total initial specification. For example the sentence:

Mick Jagger and Keith Richards write Paint It Black.

expresses an event which precedes the event expressed by the sentence:

Paint It Black is recorded, on tape number 666 in studio number 11, by The Rolling Stones.

In order to get insight of the dynamics of the UoD, each sentence of the initial specification is extended with a *time stamp.*

Resuming, an informal specification is obtained from the initial specification:

1. by adding sentences which are formulated in an active way for each passive sentence,
2. using subordinate clauses to express properties,
3. by splitting non-elementary sentences into elementary sentences, and by
4. adding a time stamp to each sentence.

Table 3.2 shows an example of a structured informal specification (with only active formulated sentences).

3.3.3.2 Analyzing informal specifications

In the sequel of the modeling process the informal specification is analyzed in such a way that the meta-model of the logbook (presented in section 4.2.1) can populated. Unfortunately we can only provide the system analyst with some rules of the thumb for grammatical analysis since this remains a difficult research topic (see remark 3.4).

Since sentences of the informal specification are explicitly time-stamped, the time component of the logbook is populated with time stamps. Sentences expressing events are broken down as much as possible into the following syntactical categories: predicate, subject, direct object, subordinate clause, and preposition phrase. The predicate specifies the action associated with the sentence. As mentioned before, subordinate clause provides the system analyst with clues for properties of objects. The other components provide the objects involved in this action, and their role (agent or goal) in the action. Notice that different types of sentences may have the same predicate. Eventually this can lead to the introduction of generalized object types (see section 4.3.2).

01-05-1963:	Mick Jagger and Keith Richards set up band 'The Rolling Stones'.
03-05-1967:	Mick Jagger and Keith Richards write Paint It Black.
21-06-1967:	The Rolling Stones record, on tape number 666 in studio number 11, Paint It Black.
23-06-1967:	Mick Jagger and Keith Richards produce the recording, on tape number 666 in studio number 11, of Paint It Black.
12-12-1988:	A. Knijff writes the song 'I Want You'.
10-02-1989:	P. Frederiks and A. Knijff set up The Playful Plebs.
29-04-1991:	The band 'Playful Plebs' record, in studio number 2 on tape number 3, the song 'I Want You'.
29-04-1991:	The band 'Playful Plebs' record, in studio number 2 on tape number 3, the song 'Long Way To Go'.
05-05-1991:	H. Honer produces the recording, recorded in studio number 2 on tape number 3, of the song 'I Want You'.

Table 3.2: Example informal specification

Remark 3.4

Ideally, a grammatical analysis of informal specifications is supported by computers. Currently many researchers in Computer Science and Linguistics are studying the possibility of applying lexicons, parsers and grammars for grammatical analysis of informal specifications[2] *(see e.g. [KO96] and [Rie94]).*

3.3.4 Abstracting the information grammar

Whenever the informal specification is analyzed into a normal form specification, the task of the analyst is to recognize the general *rules* from this normal form specification. In this step the structure of the information system is described by the conceptual model (analysis models). A type inference mechanism is applied to the sentences of the logbook such that the object types and their responsibilities, properties of the object types, and the action types and their execution order are collected. Using an enhanced type inference mechanism can lead to specialization and generalization hierarchies, and complex object types such as group types. The instances occuring in the sentences of the normal form specification are part of the (sample) population.

The normal form specification sentences are analyzed from three complementary perspectives:

1. The sentences are grouped according to their *information capacity.* Two sentences have the same information capacity if they have the same deep structure sentence.

[2]The web-site http://mu.www.media.mit.edu:80/groups/mu/ of The Machine Understanding Group contains a parser which is applicable for a large set of sentences.

Each such class corresponds with, and is labeled with, an action type, and is visualized by the so-called *object action involvement model.*

2. For each object type the sentences in which its properties are initialized or updated are grouped together. Each such class of sentences describes the *state record* of that object type. The state record for each object type is visualized by the so-called *object property model.*

3. For each object type the deep structure sentences in which that object type occurs are ordered using the time stamps in the logbook. Each such class of sentences describes the *course of life* of an object type in a generic way. The course of life of an object type is visualized by the so-called *object life model.*

These three models are also referred to as the *analysis models.* Together these models compose the conceptual model, the so-called *information architecture*, for the normal form specification. The analysis models are subject of chapters 4 and 5. Chapter 5 also provides algorithms for interpreting the analysis models into a corresponding information grammar.

Figure 3.2 shows the three perspectives on the normal form specification. An action type is denoted with A_i and an object type with O_i.

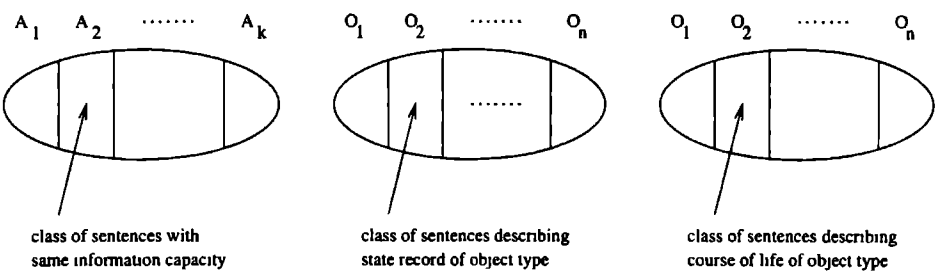


Figure 3.2: Three abstract views on the normal form specification

3.3.5 Communicating the information grammar

Once the information grammar is obtained, the analysis models are communicated to the domain experts for validation. Two ways of validating the analysis models are considered:

1. producing a textual description of the analysis models using the information grammar, and

2. obtaining a parser from the information grammar which allows the domain expert the ability to check whether sentences of the expert language are captured by the information grammar.

Chapter 6 shows a demonstration of validating an information grammar within the AGFL formalism ([Kos91]). This formalism is equipped with the so-called *grammar workbench* ([DKNZ92b]) which can be used to generate sample sentences randomly according to this information grammar. In this thesis the focus is mainly on generating typed sentences. Furthermore, the AGFL formalism contains a *parser generator* (*GEN*) which is able to generate a parser from the information grammar. Both the grammar workbench and parser generator GEN are demonstrated in chapter 6.

Remark 3.5

The quality of the information grammar G_{inf} *can be measured by:*

1. completeness:
   each sentence from the expert language can be expressed in terms of sentences from the information grammar,
2. consistency:
   the information grammar does not contain rules producing conflicting sentences.
3. naturalness:
   each sentence from the information grammar has a meaning within the UoD.

The completeness property guarantees that each expert sentence can be communicated to the information system. If the sentence is not part of the information grammar language, then the user has a formulation problem. It is expected that the formulation problem in this approach is smaller than when using 4th generation languages as SQL ([NF84]).

The information modeling procedure can now be formalized:

> An information modeling procedure M can be seen as a mapping between grammars, such that $M(G_{exp}) = G_{inf}$.

Note that M can be seen as an abstraction for the way of working as depicted in figure 3.1, starting with an expert grammar resulting in an information grammar. A first restriction, called the stabilization requirement, *is that the procedure is stable:*

$$M(M(G_{exp})) = G_{inf}$$

In other words, if the grammar G_{inf} *is subject to the analysis procedure M then the same grammar* G_{inf} *will be produced.*

3.4 Some managerial aspects

As stated in section 1.2.3 object-oriented methods claim to support reuse of classes. Furthermore, systems developed in an object-oriented way are often claimed to be cheaper and better maintainable than systems developed in a conventional way. Software project managers are often eager to take the OO plunge for these reasons, but are uncertain about the

management tasks they will face as they are unfamiliar with this new technology ([SBF96]). However, project managers should not use the novelty of the object-oriented approach as an excuse to let software developers manage themselves; they must stay involved in planning and controlling software development. Nonetheless, in many organizations developers have usurped management planning and control by convincing their managers that object-orientation is so different from other technologies that they can not understand or control it ([FC96]). The long-term results of this loss of control by project managers are almost always unfortunate, ultimately threatening the business value of the object-oriented approach ([BCG95]). As stated in [Wil96a], the single largest reason for failing of object technology projects is technology management, not the technology itself.

Obviously, the introduction of object technology requires efforts and management support from both project managers and object-oriented methods, respectively. This leads to the following questions:

1. What can project managers do to adopt object technology successfully?
2. In which way can object-oriented methods support project managers?

In [FC96], [SBF96], [SD96a], and [Wil96a] a number of guidelines and hints are provided with respect to the first question:

1. Project managers must ensure that team members have sufficient training and mentoring, as well as time to become familiar with object-oriented methods without the pressure of immediate deadlines. Because of the need to adapt the development culture to an object-oriented approach, a project manager should participate in some form of training or education on object-orientation.
2. Apply so-called *expectation management* resulting in a transition plan. The transition to object-orientation will affect for example planning aspects, budgetary aspects, and training for personnel.
3. Select the first project for applying object-oriented methods carefully. The ideal first project should be large enough to be meaningful but not larger than the organization is used to handle, and should be supported by senior management. Projects without sufficient support may fail for reasons that have nothing to do with object-orientation.
4. The development process must be documented. Adaption of object-orientation will undoubtedly change the way developers do their jobs, and a documented development process helps them to understand what is expected.
5. Use techniques to manage object-oriented development. These techniques should include a combination of milestones, and counting classes and methods (see e.g. [MN96]).

An object-oriented method (and especially a natural language based object-oriented method) can support a project manager in the following way:

1. Support the tracebility of the development process. As a result the development process can be reconstructed which has its uses. For example, there may be multiple hypotheses, false starts, experiments, and dead ends ([PG96], [Cor96]). Another advantage is that before new project members are added to a project team, they can study the history of the project. By paraphrasing the increments of the object-oriented models each iteration, the history of the project can be expressed in natural language sentences.

2. Provide education guidelines to the method to the members of project teams. Furthermore, it is useful to provide guidelines to the required skills of those involved in the project. [RP96] states that, in contrast with other occupations, we are a long way from the situation where all major companies recognize that certain activities in the development process can only be undertaken by particular, well-qualified participants in the project. Section 3.2.2 contains a number of skills for system analysts and domain experts for a natural language based modeling method. An additional advantage of these guidelines is that it provides insight in the responsibilities of the participants.

3. Deliver small intermediate products. As a result the modeling process is visible for the project manager as well as for the client, and wrong hypotheses can be detected in an early stage. The analysis models as presented in section 3.3.4 are such intermediate products.

4. Define a way to ensure the quality of the intermediate products and the final system. One way to ensure the quality is by providing so-called *validation documents* to the project manager and client which contain textual descriptions of the intermediate products and final system (see section 3.3.5).

3.5 Summary and outlook

In this chapter the natural language based modeling process introduced in chapter 1 has been further refined. For each step in the modeling process it has been stated what skills are demanded from domain expert and system analyst to perform this step. These skills have been introduced, and it has been shown that these skills intercept the drawbacks of natural language usage as a starting point for conceptual modeling. Finally, a number of managerial aspects with respect to (natural language based) object-oriented methods have been presented.

As promised in this chapter, the next chapter discusses the notion of logbooks and provides an informal introduction to the analysis models which together form a conceptual model called the information architecture.

Chapter 4

Modeling Information Grammars

Oh, a storm is threat'ning
my very life today
If I don't get some shelter
yeah I'm gonna fade away

From: "Gimme Shelter",
The Rolling Stones

4.1 Introduction

In this chapter[1] the intuition behind, and the concepts for the way of modeling and visualization of information grammars are described.

One goal of the natural language based modeling process is to obtain a conceptual model of the UoD from the normal form specification (section 3.3.3). The notion of a *logbook* is introduced in section 4.2 as such a normal form specification. The logbook is presented as a meta-model in terms of PSM ([HW93]), followed by a further elaboration of logbook functions and properties.

A logbook can be analyzed according to three complementary perspectives, leading to a number of abstractions of the logbook, also referred to as the analysis models. The intuition with respect to the concepts, graphical notation, and the integration of these three analysis models, i.e. the modeling technique, is further elaborated in section 4.3. This object-oriented modeling technique is called PSM^2 which is an extension of PSM with concepts for modeling dynamic aspects of the UoD. Finally, the results of this chapter are summarized in section 4.4.

[1] This chapter is based on parts of [BFW96], [FW96c], and [FKW96].

4.2 Logbooks

The most simple description of events occurring in a UoD consists of the following components for each event (1) *when* does it occur, (2) *what* is happening, (3) *who* are associated, (4) what is the *role* of each associate and (5) which *properties* of the associates are relevant A *logbook* is a sequence of such event descriptions, ordered by their time stamps in some unifying format (seconds, hours, days, etc)

The starting-point for constructing a logbook is an informal specification consisting of natural language sentences describing time-stamped events Obtaining an informal specification is usually an incremental and iterative process ([RP96], chapter 3) which requires skills from both the domain expert and system analyst (chapter 3) As an example of an informal specification consider table 3 2 in section 3 3 3 where the unifying time format is chosen to be on a daily basis

4.2.1 Meta-model of logbooks

Formally, the format of a logbook is defined by the meta-model of figure 4 1. This meta-model is presented as a PSM schema which is a formalized and extended version of the object-role modeling technique NIAM ([Hal95] or [NH89]) For more literature about the background and formalization of PSM, the reader is referred to [HW93] or [Hof93]

A PSM *schema* is a structure consisting of an *information structure* and a set of *constraints* The semantics of a PSM schema is the set of its possible *populations*, which is restricted by both the information structure and the constraints

The information structure consists of the following basic components

1 A set of *predicators* A predicator is intended to specify the role played by an object type in a fact type (see below)

2 A set of *object types* which can be classified as follows

   (a) *Entity types* and *label types* The difference is that labels can, in contrast with entities, be represented (reproduced) on a communication medium

   (b) *Fact types* The set of fact types is a partition of the set of predicators

   (c) Complex object types such as *power types* (also called *set types*), *sequence types* (for modeling list structures), and *schema types* (which are used as an abstraction mechanism for information structures)

3 Relations for specifying (a) *specialization* and (b) *generalization*

4 Functions for specifying (a) the object type associated to a predicator, (b) the fact type of an associated predicator, and (c) the underlying element type (or object type) for power types and sequence types (and schema types)

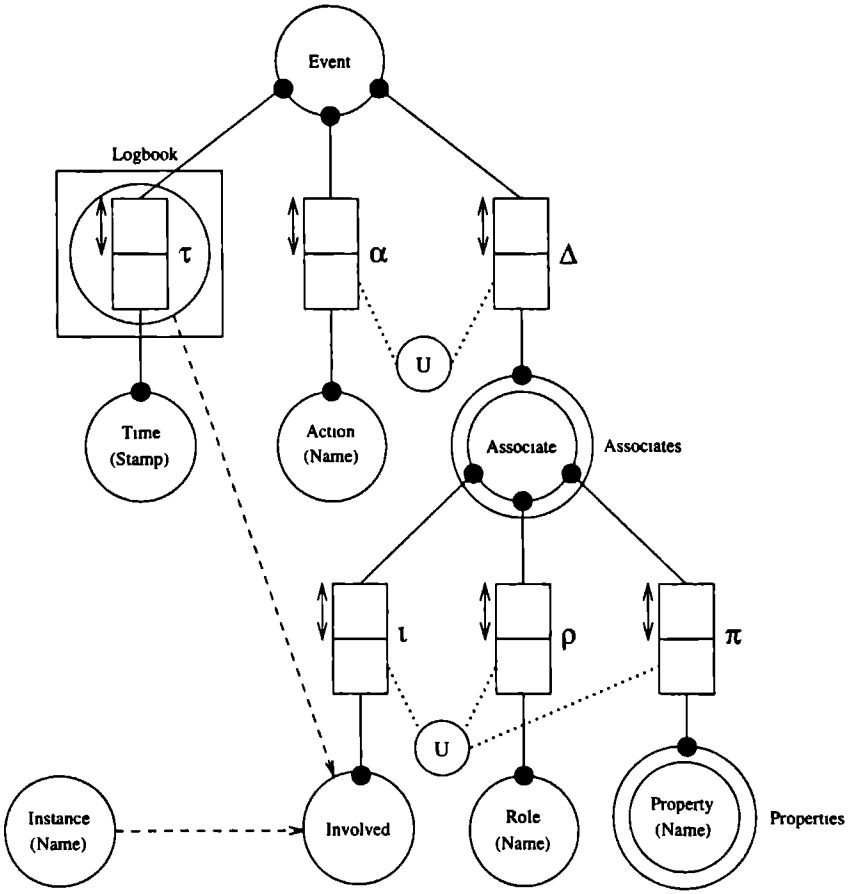


Figure 4.1: Meta-model for the format of a logbook

In figure 4.1 we have a number of entity types, e.g. *Event*, represented by circles. Label types, also represented by circles, appear between parentheses. Examples of label types are *Name* and *Stamp*[2]. The convention is used that if a label type is an identifier for an entity type, the label type is represented within the same circle. For example, in the logbook meta-model an *Action* is identified by its *Name*. Fact types consist of predicators represented by boxes and connected to a circle for their associated object type. For example, fact type τ is a binary fact type consisting of two predicators, with associated object types *Event* and *Time*. Object type *Properties* is modeled as a power type with underlying element type *Property*. A *Logbook* is viewed as a sequence type of tuples consisting of an *Event*

[2]The population of label type *Stamp* is assumed to be an ordered set. Furthermore, time stamps are assumed to be discrete.

and a *Time* part. Finally, the information structure of a logbook contains a generalization hierarchy. Object type *Involved* is either an *Instance* or an *Event/Time* tuple.

The PSM schema of figure 4.1 also contains a number of constraints. A *total role* constraint, denoted by a black dot, expresses that each instance of the corresponding object type must be involved in the corresponding fact type. For example, each *Event* must have a *Time* component. The arrows and 'circles with an *U*' represent so-called *uniqueness* constraints over a predicator and a set of predicators, respectively. The uniqueness constraint is used to specify that each instance of the corresponding object type is involved *only once* in the corresponding fact type *provided* it plays that role. Note that both a total role and a uniqueness constraint over a predicator states that each instance of the corresponding object type is involved *exactly once* in the corresponding fact type. Thus such a predicator is practically a function. In the sequel of this section we use this observation and name this function by the name of its corresponding fact type.

In order to be able to represent informal specifications in a way as presented in table 3.2, some additional (non-graphical) constraints are required. The PSM modeling technique is equipped with the query language Lisa-D to express such constraints. Lisa-D as such falls outside the scope of this thesis, for more information see [HPW93]. The additional constraints are informally expressed as:

1. the order of the tuples of a logbook is according to the order of its time stamps, and
2. if two events occur at the same time, these events can not be the same.

Table B.1 (given in appendix B) represents a population of the meta-model of the logbook in figure 4.1 and is thus a unifying format for the sample informal specification of table 3.2. Note that the action *to write* may have multiple agents. However, each of these events is elementary, and thus can not be split in smaller events. As a consequence, the modeling technique must be able to handle the case of multiple agents on a conceptual level.

4.2.2 Logbook functions

In this section a number of functions are defined in order to elaborate the notion of a logbook as defined by its meta-model. The function $\tau(e)$ is such a function which produces the point of time of an event e. $\alpha(e)$ results in the performed action and $\Delta(e)$ is the mapping which assigns the set of associates occuring in event e.

Let a be an associate of event e, i.e., $a \in \Delta(e)$. The function $\iota(a)$ yields the involved object, $\pi(a)$ assigns the (possibly empty) set of properties of this object, whereas $\rho(a)$ provides its role in this event. Note that a role may be played more than once in an event.

An event e thus is identified by the action $\alpha(e)$ performed, and the set $\Delta(e)$ of associates. An associate a is identified by its involved object $\iota(a)$, its corresponding set of properties $\pi(a)$, and its role $\rho(a)$.

We will use $\iota \circ \Delta$ as an abbreviation for:

$$\langle e, x \rangle \in \iota \circ \Delta \iff \exists_{A, a \in A} [\langle e, A \rangle \in \Delta \wedge \langle a, x \rangle \in \iota]$$

Furthermore, we use $\iota \circ \Delta(e)$ to denote the set of all objects involved in event e. $\rho \circ \Delta$ and $\pi \circ \Delta$ are defined analogously.

Example 4.1

Consider the event e_1 with time stamp (*23-06-1967*) from table B.1:

$$\left\langle \text{produce}, \left\{ \begin{array}{l} \left\langle e_2, \text{object}, \left\{ \text{tape number 666}, \text{studio number 11} \right\} \right\rangle, \\ \langle \text{Mick Jagger}, \text{agent}, \varnothing \rangle, \\ \langle \text{Keith Richards}, \text{agent}, \varnothing \rangle \end{array} \right\} \right\rangle$$

and the event e_2 with time stamp (*21-06-1967*):

$$\left\langle \text{record}, \left\{ \begin{array}{l} \langle \text{Paint It Black}, \text{object}, \varnothing \rangle, \\ \langle \text{The Rolling Stones}, \text{agent}, \varnothing \rangle \end{array} \right\} \right\rangle$$

which events correspond with the following sentences of the informal specification:

> *Mick Jagger and Keith Richards produce the recording, on tape number 666 in studio number 11, of Paint It Black.*
>
> *The Rolling Stones record, on tape number 666 in studio number 11, Paint It Black.*

Applying the functions τ, α, Δ, ι, ρ, and π to events e_1 and e_2 results in:

$\tau(e_1)$	=	23-06-1967	$\tau(e_2)$	=	21-06-1967
$\alpha(e_1)$	=	produce	$\alpha(e_2)$	=	record
$\Delta(e_1)$	=	$\{a_1, a_2, a_3\}$	$\Delta(e_2)$	=	$\{a_4, a_5\}$
$\iota(a_1)$	=	e_2	$\iota(a_4)$	=	Paint It Black
$\iota(a_2)$	=	Mick Jagger	$\iota(a_5)$	=	The Rolling Stones
$\iota(a_3)$	=	Keith Richards			
$\rho(a_1)$	=	object	$\rho(a_4)$	=	object
$\rho(a_2)$	=	agent	$\rho(a_5)$	=	agent
$\rho(a_3)$	=	agent			
$\pi(a_1)$	=	$\left\{ \begin{array}{l} \text{tape number 666}, \\ \text{studio number 11} \end{array} \right\}$			
$\pi(a_2)$	=	$\varnothing$	$\pi(a_4)$	=	$\varnothing$
$\pi(a_3)$	=	$\varnothing$	$\pi(a_5)$	=	$\varnothing$

4.2.3 Properties of logbooks

In this approach, a concrete logbook is an instantiation of the meta-model from figure 4.1, i.e. a mapping which instantiates each object type. Let Log be such a mapping (population). Function application to the object types of the logbook meta-model leads to some useful results. For example, the set of all involved objects occurring in this logbook can be obtained with Log(Involved). This set corresponds to what is called the *extra-temporal population* in [PW95a].

The logbook can be restricted to the history (evolution) Log_x of a single involved object x by removing all events from the logbook in which x is not involved. The resulting population of the meta-model is obtained by first determining the relevant events:

$$\mathsf{Log}_x(\tau) = \left\{ \langle t, e\rangle \in \tau \;\middle|\; x \in (\iota \circ \Delta)(e) \right\}$$

where for convenience $\langle t, e\rangle \in \tau$ is used as a shorthand for $\langle t, e\rangle \in \mathsf{Log}(\tau)$ (thus overloading the $\in$-operator). The instantiations of the other object types of the meta-model are obtained as the minimal subset of their original instantiation required for $\mathsf{Log}_x(\tau)$.

Restricting a logbook Log to a particular point of time (time stamp) t results analogously in a *snapshot* of the logbook:

$$\mathsf{Log}_t(\tau) = \left\{ \langle t, e\rangle \in \tau \;\middle|\; \tau(e) = t \right\}$$

The history of a single action a is obtained from:

$$\mathsf{Log}_a(\tau) = \left\{ \langle t, e\rangle \in \tau \;\middle|\; \alpha(e) = a \right\}$$

Note that $\mathsf{Log}_a(\mathsf{Involved})$ results in the set of objects involved in action a.

Many object-oriented modeling methods involve one or more models describing the life of objects. Especially the creation (birth) of an object has a prominent position in these life models. The creation $\mathsf{Init}(x)$ of an object x is defined as the first event in the logbook which mentions this object. We assume that an object can be involved in at most one action at each moment.

$$\mathsf{Init}(x) \in \mathsf{Log}_x(\mathsf{Event}) \wedge \forall_{e \in \mathsf{Log}_x(\mathsf{Event})} \left[\tau(\mathsf{Init}(x)) \leq \tau(e)\right]$$

As a result $\alpha(\mathsf{Init}(x))$ denotes the action which causes the birth of object x. Note that the death of an object can not be derived from the logbook since objects are not removed from the logbook. Usually there will be a criterion $\mathsf{Alive}(x, t)$ which decides whether object x is alive at some point of time t. The following property then is obvious:

$$\mathsf{Alive}(x, t) \wedge \forall_{s<t} \left[\neg\, \mathsf{Alive}(x, s)\right] \Rightarrow \tau(\mathsf{Init}(x)) = t$$

4.2.4 Views on logbooks

During the several stages of the modeling process the sentences of a logbook are analyzed according to three perspectives, leading to a number of abstractions of the logbook, also

referred to as the *analysis models*. Each abstraction provides a specific view on the nature of the application domain, and results in a corresponding model. The models together compose a conceptual model of the UoD. This conceptual model is also referred to as the *information architecture*. The information architecture composes the following analysis models:

- *object action involvement model*,
- *object property model*, and
- *object life model*.

The purpose of the *object action involvement model* is to develop a first approximation to the information grammar, covering the main structure of the logbook language. This model provides an abstraction (typing) for three columns of the logbook, i.e. *Action*, *Involved*, and *Role* (see table B.1). Special is that different types of events may have the same type of action. Eventually this can lead to the introduction of generalized object types (see section 4.3.2).

The *object property model* deals with static aspects of the application domain, by modeling *properties* of object types. The properties of an object type form a so-called *state record* which reports about the current state of an object. Usually properties are set during the birth of an instance of the associated object type. Properties are only useful if they can be retrieved via *retrieval* action types and updated by so-called *update* action types. Strictly spoken, properties thus can also be modeled via its initialization action type, retrieval action types and update action types. The object property model is introduced to provide a more simple description mechanism dealing with properties. This model is obtained by inter alia consulting the *Property* column of the logbook.

The object action involvement model does not restrict the order of the action types. The *object life model* elaborates on the object action involvement model. The object life model considers the application domain from a historical perspective, and describes for each object type the *course of life* of its instances. Each object type starts with its birth action type (creation). From the object action involvement model it can be derived in which action types an object type can be involved. Using the column *Time* of the logbook, clues for the ordering of action types can be derived. Note that the object life model describes a possible order in which action types can be performed for each object type. By collecting all life courses the (restrictions on the) dynamics of the UoD can be derived.

4.2.5 Typing logbooks

In order to obtain structural information from a logbook, all instances that occur in the sentences (tuples) of the logbook have to be typed by the system analyst according to some *type inference mechanism*. This type inference mechanism is guided by the three analysis models that constitute the information grammar. This stage of the modeling process may informally be described as follows:

Find a consistent type inference mechanism, i.e. a type inference mechanism satisfying:

1. *Events with a similar action have (a) similar objects involved in (b) similar roles.*
2. *Similar objects have a similar state record.*
3. *Similar objects have a similar course of life.*

Two instances of the logbook meta-model are said to be *similar* if they have been assigned the same type to. Two sets of instances are called similar if each instance in the first set has a similar instance in the second set, and vice versa. Recognizing similar lives of objects is hard to formalize. In fact this remains the main task of the system analyst: ordering and comparing the course of life of objects. In any case, two objects are candidate for similar life if they have a similar birth action.

Based on this intuition the type inference mechanism can be formally stated as follows: find a set Types of types and a typing function

$$\mathsf{Type} : \mathsf{Log}(\mathrm{Involved}) \cup \mathsf{Log}(\mathrm{Action}) \cup \mathsf{Log}(\mathrm{Role}) \cup \mathsf{Log}(\mathrm{Property}) \rightarrow \mathsf{Types}$$

such that:

1. Events e_1 and e_2 with a similar action have (a) similar objects involved (b) in similar roles:

$$\mathsf{IsSim}(\alpha(e_1), \alpha(e_2)) \Rightarrow \mathsf{IsSim}(\iota \circ \Delta(e_1), \iota \circ \Delta(e_2)) \wedge \mathsf{IsSim}(\rho \circ \Delta(e_1), \rho \circ \Delta(e_2))$$

2. Similar objects x and y have a similar state record:

$$\mathsf{IsSim}(x, y) \Rightarrow \mathsf{IsSim}(\mathsf{Log}_x(\mathrm{Properties}), \mathsf{Log}_y(\mathrm{Properties}))$$

3. Similar objects x and y have a similar course of life:

$$\mathsf{IsSim}(x, y) \Rightarrow \mathsf{SimLife}(x, y)$$

The similarity predicate is defined on instances x and y as:

$$\mathsf{IsSim}(x, y) \Leftrightarrow \mathsf{Type}(x) = \mathsf{Type}(y)$$

whereas on sets X and Y of instances:

$$\mathsf{IsSim}(X, Y) \Leftrightarrow \forall_{x \in X} \exists_{y \in Y} [\mathsf{IsSim}(x, y)] \wedge \forall_{y \in Y} \exists_{x \in X} [\mathsf{IsSim}(y, x)]$$

Note that this definition does not require similar sets of instances to have equal cardinality. Similarity on life of objects is defined as:

$$\mathsf{SimLife}(x, y) \Leftrightarrow \mathsf{IsSim}(\alpha(\mathsf{Init}(x)), \alpha(\mathsf{Init}(y)))$$

The similarity predicate forms an equivalence relation over the instances of the logbook.

This type inference mechanism is very strict. The type inference mechanism can be enhanced using the *type relatedness* predicate (see chapter 5), which extends the type inference mechanism with a subtyping mechanism (specialization and generalization). For example, in some UoD regarding bands and musicians, different object types, such as *Person* and *Musician*, can share common instances. A *Musician* might be modeled in this UoD as a specialization of a *Person*. In this UoD, *Person* and *Musician* are type related but not of similar type. Introduction of a subtyping mechanism requires a refinement of the type inference mechanism. For example, similar objects within the same subtype hierarchy need not to have similar state records, e.g. object type *Musician* can have properties which are not relevant for object type *Person*.

Example 4.2

Suppose the type inference mechanism yields the following:

$$\begin{aligned} \mathsf{Type}(\textit{Mick Jagger}) &= \textit{Person} \\ \mathsf{Type}(\textit{The Rolling Stones}) &= \textit{Person} \end{aligned}$$

Then we have no consistent typing, as Mick Jagger and The Rolling Stones do not have a similar life.

Example 4.3

Suppose the type inference mechanism yields the following:

$$\begin{aligned} \mathsf{Type}(\textit{Mick Jagger}) &= \textit{Person 1} \\ \mathsf{Type}(\textit{Keith Richards}) &= \textit{Person 2} \end{aligned}$$

This typing is not plausible, as Mick Jagger and Keith Richards occur in events with similar actions in similar roles.

4.3 Analysis models

This section introduces the analysis models and its concepts informally. Before we discuss the analysis models by examples, the concepts used are introduced. Furthermore, two important principles for conceptual modeling are discussed.

4.3.1 Requirements and concepts

In order to model a UoD adequately, a system analyst must understand the problem domain precisely. This requirement is stated by the so-called *Conceptualization Principle* and the *100 Percent Principle* ([Gri82]). The Conceptualization Principle states that conceptual models should deal only and exclusively with aspects of the Universe of Discourse. The 100 Percent Principle states that conceptual models should describe all dynamic and static

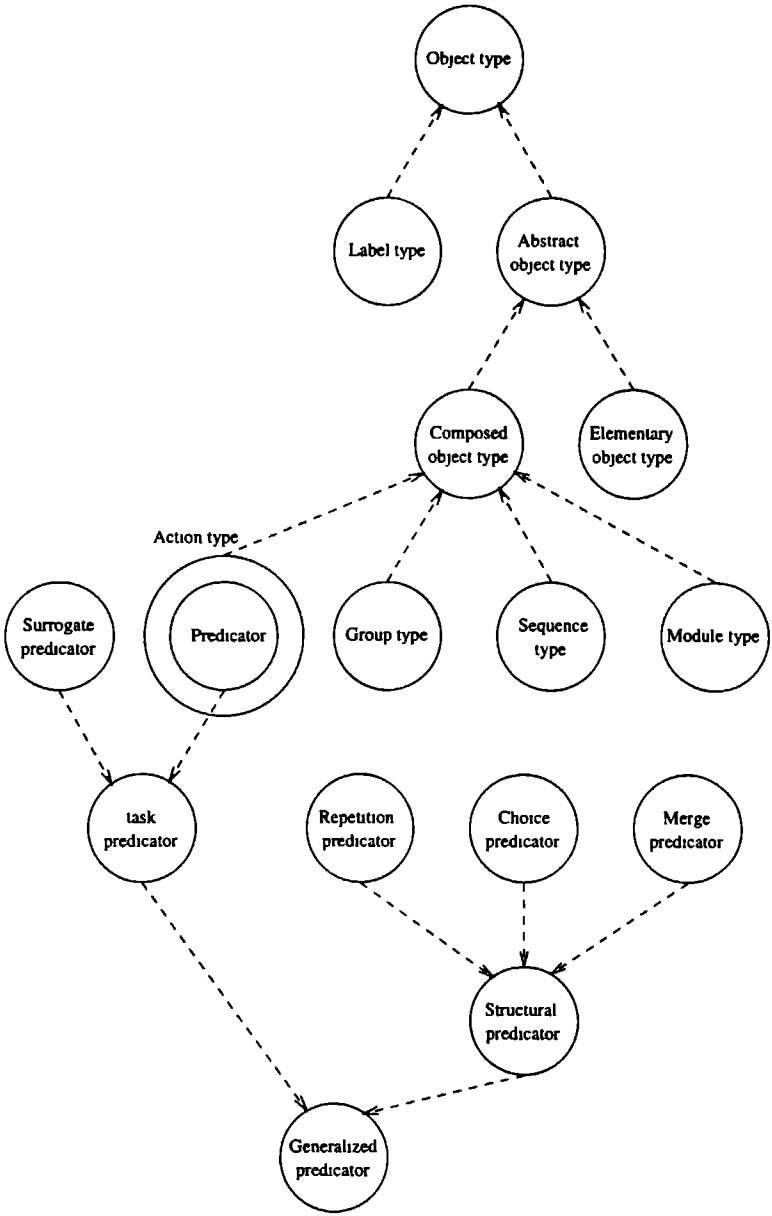


Figure 4.2: Inclusion relation of object types and generalized predicators

aspects of the Universe of Discourse. As a consequence, the conceptual modeling technique used by the system analyst has to provide the system analyst with the proper concepts.

The concepts introduced in the sequel of this section are strongly related to the concepts of PSM (section 4.2.1). As PSM is a *data* modeling technique, some of these concepts are renamed. Furthermore, the concept of predicator is extended in order to express dynamic aspects of a UoD. This extended notion of predicator is called a *generalized* predicator or *component*. The set of general predicators can be divided into a set of *structural* predicators and a set of *tasks* (task predicators). The structural predicators refer to control structures such as *repetition* predicators, *choice* predicators, and *merge* predicators. A task is either a *predicator* or a *surrogate* predicator (also called *deputy* predicator). Predicators are marked with either an r or an i, stating whether the object type connected with that predicator is *responsible* for, or *involved* in the execution of the action type associated to that predicator. The PSM meta-model of figure 4.2 shows the inclusion relation with respect to the object types and generalized predicators. Table 4.1 provides examples (in the context of bands and the music industry) of the different types of object types.

Kind of object type	example
Label type	Number (to denote a tape number), Name (to denote a band name)
Elementary object type	Person, Song
Action type	to write (person writes song)
Group type	Band (a band is a group of musicians)
Sequence type	CD (a CD is a sequence of songs)
Module type	Music industry (the music industry consists of companies such as EMI and Sony)

Table 4.1: Examples of object types

Table 4.2 provides an overview of the concepts of the analysis models and its counterparts (if existing) in PSM. The analysis models can be seen as an extension of PSM with concepts for modeling dynamic aspects. As a result, the two dimensions of a UoD (static and dynamic dimension) can both be represented with the analysis models. This is also emphasized in a new name for the modeling technique as presented in this thesis: PSM² (PSM square).

Finally, we like to stress that the semantics of a PSM model differs from the semantics of PSM². The semantics of a PSM model is set theoretically based. PSM² is used to model the logbook. As a consequence, for the semantics of PSM² the notion of time stamps plays a dominant role. The semantics of the object action involvement model and the object property model is based on set theory extended with the notion of time (see also [TL91]). Process algebra and a theory for traces are used for the semantics of object life models.

In the next sections PSM² is further explained using examples. A formal treatment is given

PSM2	PSM
predicator	predicator
object type	object type
elementary object type	entity type
label type	label type
action type	fact type
group type	power type
sequence type	sequence type
module type	schema type
specialization	specialization
generalization	generalization
properties	-
triggers	-
surrogate predicators	-
repetition predicators	-
choice predicators	-
merge predicators	-

Table 4.2: PSM2 concepts versus PSM concepts

in chapter 5. An overview of the graphical conventions for the models of PSM2 can be found in appendix C.

Remark 4.1

Recently, more extensions of object-role data modeling techniques to object-oriented modeling techniques have been proposed in [CP96] and [De 96]. These approaches focus on the concepts of the way of modeling.

4.3.2 Object action involvement model

The object action involvement model provides a structured description of the structure of events in the UoD stating which (abstract) object types are involved in what action types and in what role. A special kind of involvement indicates which object type(s) is (are) responsible for what action type. These object types thus are seen as the initiators of that action type. External initiators of action types, so-called *subject types*, fall outside the scope of this thesis. The model may also contain composed object types such as generalized object types and group types.

As an example consider the following two events from the logbook of table B.1:

$$\left\langle \text{record}, \left\{ \begin{array}{l} \langle \text{The Rolling Stones}, \text{agent}, \varnothing \rangle, \\ \langle \text{Paint It Black}, \text{object}, \varnothing \rangle \end{array} \right\} \right\rangle \left\langle \text{record}, \left\{ \begin{array}{l} \langle \text{The Playful Plebs}, \text{agent}, \varnothing \rangle, \\ \langle \text{I Want You}, \text{object}, \varnothing \rangle \end{array} \right\} \right\rangle$$

These sentences lead to the recognition of (1) the object types *Band* and *Song*, and (2) the action type *to record*. Let *ag* and *ob* denote the respective roles of *Band* and *Song* in this action type. Then action type *to record* is verbalized as:

ag records *ob*

Another association could be:

ob is recorded by *ag*

Another example are the sentences concerning the production of recordings. The logbook states that both *H. Honer* and *The Rolling Stones* produce recordings. Assuming that *H. Honer* is a person and *The Rolling Stones* is a band, we obtain two action types both named *to produce* and both verbalized as:

ag produces *ob*

where *ag* denotes the role played by either *Person* or *Band* in action type *to produce*, and *ob* the role played by *Recording*. Recognizing that these associations have the same predicate, we introduce the generalized object type *Producer*. Figure 4.3 resumes the discussion above. Furthermore, the model contains an action type *to write*, involving persons and songs. Note that object type *Recording* is the objectification of action type *to record*.

4.3.3 Object property model

Properties bridge the gap between abstract (elementary or composed) object types and concrete (label) types. The properties of an object type form a *state record* (see section 3.3.4) which reports about the current state of an object. Usually the state record is initiated whenever an object is created (its birth). During the life of an object its state record may be inspected by so-called *retrieval* actions, and updated by so-called *update* actions. Strictly spoken, a property can be modeled by its initialization action, retrieval actions, and update actions, and thus be included in the object action involvement model and object life model. However, such an inclusion will lead to complex models with no clear distinction between the levels of abstraction. The object property model provides a more convenient description for properties of object types.

Obviously the object types in figure 4.3 have properties. For example, instances of object type *Band* have a name. This property (of having a name) is initialized whenever a band is set up. During the life of a band its band name may be inspected via retrieval actions. In this particular example there are no explicit update action types for band names, except for its initialization (which can be seen as an update). Figure 4.4 shows an example of an object property model where *(Name)* denotes a concrete object type. The *trigger* relation, which is a relation between an action type and an update action type, states that action type *to set up* causes an update of property *Band name*. The dashed box specifies which update

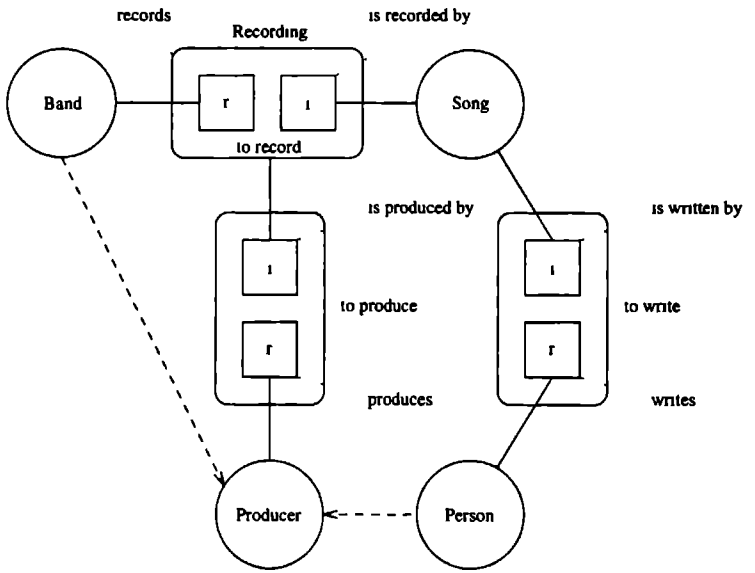


Figure 4.3: An object action involvement model

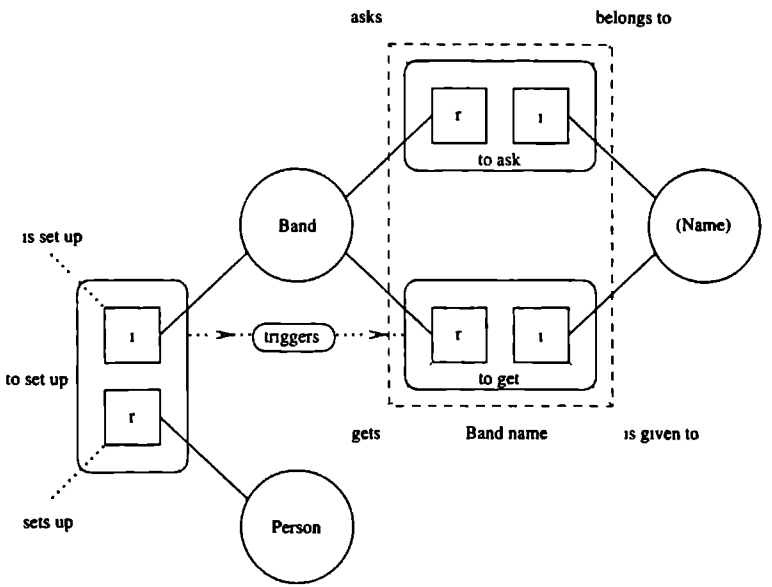


Figure 4.4: An object property model

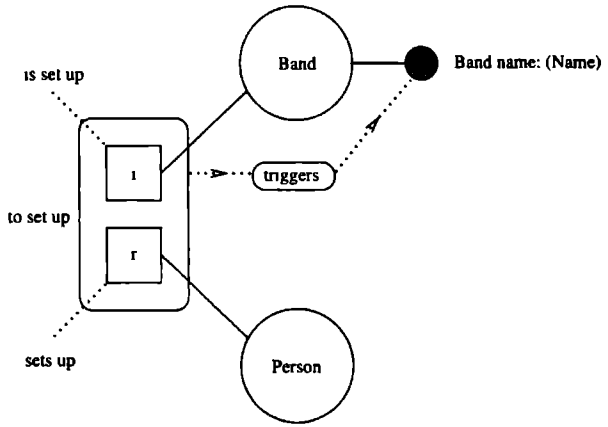


Figure 4.5: Shorthand notation

action type and retrieval action type belong to the property *Band name*. The shorthand graphical representation for the object property model of figure 4.4 is given in figure 4.5.

The object property models of figures 4.4 and 4.5 require the following additional constraints:

1. When a band is set up it also gets a band name.
2. A band name belongs to a band only when it is given to that band.

This can be generalized as follows:

1. Modifications are a side-effect of some action type. They can thus be represented by a trigger mechanism.
2. A property can only be retrieved after it has been set.

The property model describes what properties object types have and which action types trigger their modifications. The execution order of action types (and thus also retrieval- and update action types) are considered in the object life model.

4.3.4 Object life model

The object action involvement model states which object types are involved in what action type. However, this model does not state the order in which actions may be performed. The object property model provides only an ordering (the trigger relation) between action types, which cause an update of the state record of an object type, and their corresponding update action types.

Object life models elaborate on both the object action involvement model and the object property model. The goal of the object life model is to describe the so-called *course of life* for each object type. The course of life of an object type describes the set of possible sequences in which action types (including retrieval action types and update action types) may be invoked subsequently. Such a sequence is also referred to as a *(process) trace* and describes the behavior or *histories* of individual objects of that type. The first action in this sequence is called the *initialization* or *birth action*.

Reconsider the figures 4.3 and 4.4. The object action involvement model states that (1) a band is set up by a person, (2) a band can record songs and (3) a band can produce recordings during its life. We also assume bands to be able to participate in the following actions:

1. a band is joined by a person,
2. a band is left by a person, and
3. a band disbands.

Furthermore, the object property model states that a band has the property *Band name*.

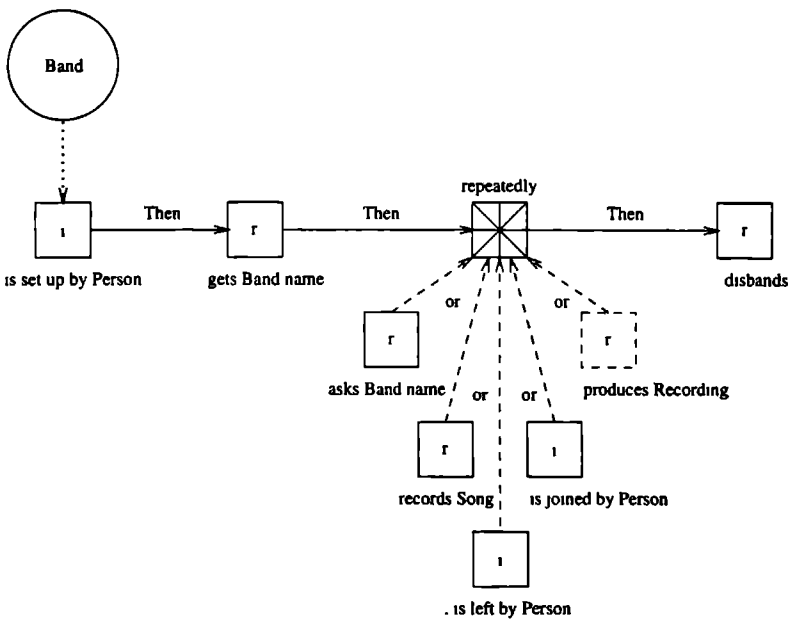


Figure 4.6: An object life model including retrieval actions

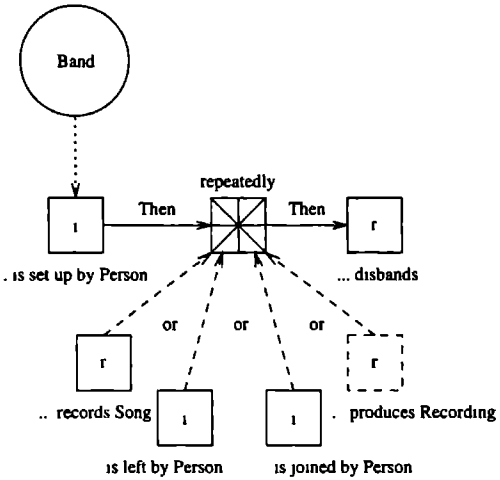


Figure 4.7: An object life model

Figure 4.6 shows the life cycle of object type *Band*. This model expresses that an object of type band is born when it is set up by a person; after that the band gets its name. Then a band can repeatedly (zero or more times): be joined or left by a person, record a song, produce a recording, and ask for its name via its retrieval action type. Finally, the life of a band is (explicitly) ended after it disbands. Note that additional constraints are necessary to exclude histories where:

1. a person leaves a band before joining that band,
2. a person joins a band already having that person as a member.

Such restrictions on histories can not be expressed by the object life model graphically and will be the subject of chapter 5.

The dotted arrow in figure 4.7 connects the corresponding object type with its course of life via its birth action. The course of life consists of generalized predicators which are mutually connected by either a solid arrow or a dashed arrow. The solid arrows denote the sequential order of generalized predicators whereas the dashed arrows denote a decomposition relation between generalized predicators and structural predicators. The predicators used in the object life model are those predicators which have already appeared in the object action involvement and object property model having the corresponding object type as actor, i.e. an object type connected with that predicator. Surrogates, denoted with dashed boxes, refer to predicators which appear in a generalization or specialization hierarchy. Furthermore, surrogates are used to express multiple occurrences of predicators in the course of life of an object type. The symbols which can be used for an object life model are summarized in appendix C.

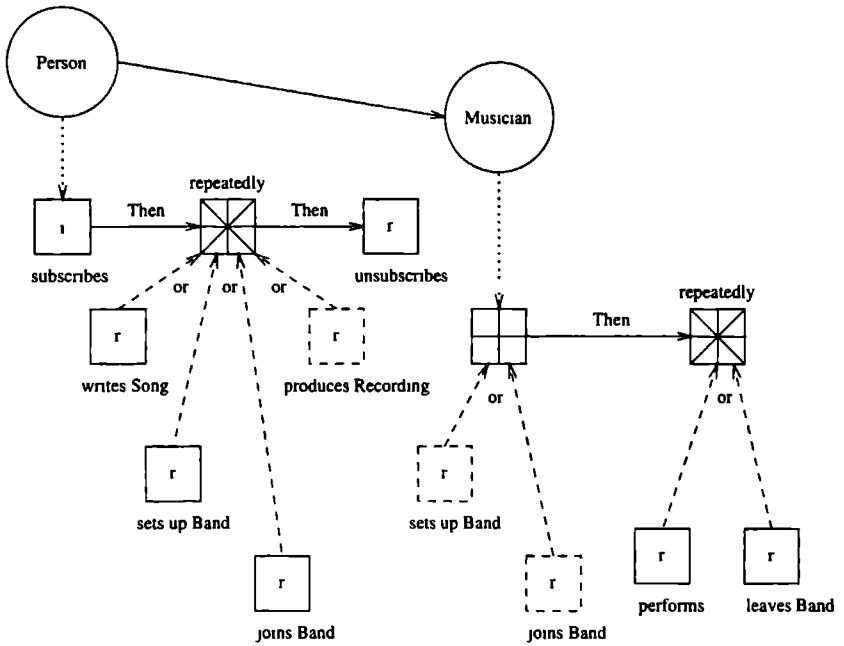


Figure 4.8: An object life model including specialization

Next we focus on the relation between the course of life of an object type and on that of its *subtypes*, and the relation between the course of life of a generalized object type and that of its *specifiers*.

The course of life of a *supertype* is included in the courses of life of its subtypes. On the other hand, the life of a subtype may contain actions from its supertypes. Assume, in the context of a music company, that object type *Person* is initialized by subscribing and ended by unsubscribing. During his/her life time a person can write songs, produce recordings, set up bands, and join bands. When an object of type *Person* sets up a band or joins a band it becomes also an instance of type *Musician*. A musician remains a person, and thus musicians can do the things persons do. But musicians can do more. For example, only a musician can leave bands and give performances. Figure 4.8 shows the object life model of a person and musician. The solid arrow between *Person* and *Musician* denotes the specialization relation.

The concept of generalization is used to model parts of the courses of life which the *specifiers* have in common. The course of life of a generalized object type contains tasks which are part of the course of life of *each* specifier. An instance of a specifier is by definition an instance of its generalized object type with the potention to be involved in the action types of the generalized object type. The object life model of a specifier states at which point in its course of life an instance is allowed to be involved in the action types of the generalized

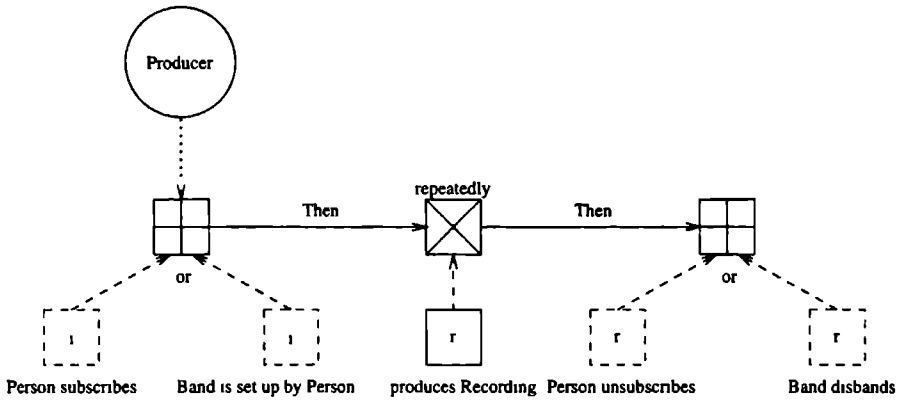


Figure 4.9: The course of life of a generalized object type

object type.

In our running example we have one generalized object type, i.e. *Producer* with specifiers *Band* and *Person* which have action type *to produce* in common. The object life model of figure 4.9 describes the course of life of a producer. Note that the task '... *produces Recording*' in the object life model of *Producer* is a part of the object life models of its specifiers *Band* and *Person*. Obviously, the concept of generalization supports the concept of *polymorphism*, i.e. the principle that one same action type may have different involved object types depending on the context.

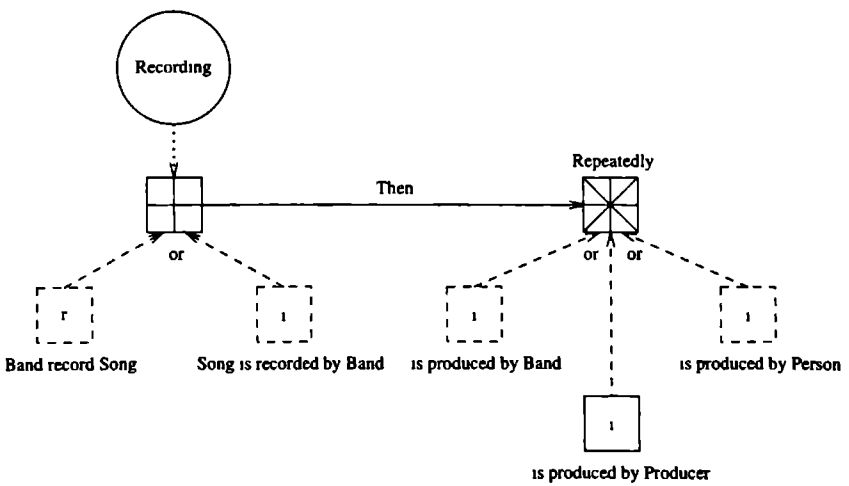


Figure 4.10: Course of life of an objectified action type

Objectified action types, i.e. action types which are also involved in other action types, also have a course of life. Except for the birth action the course of life of an objectified action type is treated in the same way as the course of life of a regular object type. An objectified action type comes alive when the corresponding action type is executed. Figure 4.10 describes the life of objectified action type *Recording* which is an objectification of action type *to record*. An instance of *Recording* comes alive when a band records a song. After a recording is made it can be produced.

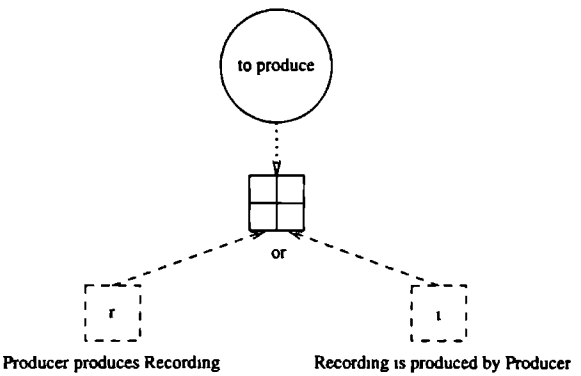


Figure 4.11: Course of life of a non-objectified action type

It is also possible to model the life of non-objectified action types. However, the course of life of such action types is redundant since it can be deduced from the object action involvement model directly. The course of life of an instance of a non-objectified action type starts when one of its predicators is executed. Consider as an example action type *to produce*. This action type is not objectified. Its course of life is shown in figure 4.11. Though, an objectification of action type *to produce*, i.e. *Production*, could be a meaningful object type in our sample UoD.

Finally, object life models for label types are discussed. Label types correspond with concrete types such as integers and are only involved in retrieval action types and update action types. Usually, label types are involved in more than one property of more than one object type. The predicators of all retrieval and update action types in which a label type is involved are enclosed in the course of life of that label type.

4.4 Summary and outlook

In this chapter the notion of a logbook has been discussed and chosen as a candidate normal form specification. A logbook can be analyzed according to three complementary perspectives, leading to a number of abstractions of the logbook, also referred to as PSM2 or the analysis models. As the name suggests, PSM2 has been inspired by the concepts of the

data modeling technique PSM. PSM2 has been obtained by extending PSM with concepts for process modeling, and changing the underlying semantics in such a way that it can model logbooks.

In this chapter PSM2 has been informally introduced by examples. In chapter 5 a formal treatment of PSM2 is provided.

Chapter 5

Formalizing Information Grammars

You better stop look around
Here it comes
Here it comes your nineteenth nervous breakdown

From: "19th Nervous Breakdown",
The Rolling Stones

5.1 Introduction

This chapter[1] provides a formal basis for the concepts and analysis models introduced in chapter 4, i.e. a formal basis for the way of modeling and validating (see section 2.3).

For each analysis model a formal syntax and semantics is described. Furthermore, an algorithm to obtain the information grammar of these analysis models is provided. Together these models form a conceptual model, called the *information architecture*, describing the structure of the communication of the UoD, i.e. the logbook, which is an integrated system with respect to (1) the graphical notations used, (2) the formal foundations, and (3) the composition of the rules of the underlying information grammar.

First the object action involvement model will be presented in section 5.2. In section 5.3 the object property model is discussed. The object life model is the subject of section 5.4. Finally, a summary of this chapter and an outlook to the next chapters is provided in section 5.5.

5.2 Object action involvement model

In this section formal aspects (syntax and semantics) of the object action involvement model are discussed. Furthermore, an algorithm to obtain a part of the information grammar (that part which is described by the object action involvement model) is described.

[1]This chapter is based on [FW96c].

5.2.1 Syntax

Formally, an *object action involvement model* is a structure $\mathcal{OAI}$ consisting of the following basic aspects:

1. A set $\mathcal{O}$ of *object types*. The set of object types is the disjoint union of the set $\mathcal{E}$ of *elementary* object types ($\mathcal{E} \subseteq \mathcal{O}$) and the set $\mathcal{Q}$ of *complex* or *composed* object types ($\mathcal{Q} \subseteq \mathcal{O}$).

2. A set $\mathcal{P}$ of *predicators*. Predicators are abstractions of roles in actions. Predicators are characterized by the function χ. If $\chi(p) = r$, then the object playing this role is characterized as being responsible for this action. The set Γ of characteristics for predicators includes $\{r, i\}$.

3. An *action type* is seen as the set of its predicators. As a result, the set $\mathcal{A}$ of action types forms a partition of the set $\mathcal{P}$. Action types are composed object types ($\mathcal{A} \subseteq \mathcal{Q}$).

   The action type associated with an predicator p is denoted by $\mathsf{Action}(p)$, while $\mathsf{Actor}(p)$ denotes the object type being its actor. We use the notation $p \in a$ as a shorthand for $\mathsf{Action}(p) = a$.

4. A set $\mathcal{G}$ of *group types*. Group types form a special class of composed object types, i.e. $\mathcal{G} \subseteq \mathcal{Q}$. Group types are used for example to handle multiple roles of an object type in an action type.

5. A set $\mathcal{S}$ of *sequence types*. Sequence types form a special class of composed object types, i.e. $\mathcal{S} \subseteq \mathcal{Q}$. Sequence types are used to express ordered lists.

6. The function $\mathsf{Elt} : \mathcal{G} \cup \mathcal{S} \rightarrow \mathcal{O}$ yields the underlying *element type* of group types and sequence types.

7. A set $\mathcal{M}$ of *module types* ($\mathcal{M} \subseteq \mathcal{Q}$). Module types can be used to express model decompositions, which is especially useful in the case of large application domains. The relation $\mathsf{compr} \subseteq \mathcal{M} \times \mathcal{O}$ describes how a module type is decomposed into its components, where $x \,\mathsf{compr}\, y$ expresses that y is part of module x.

8. The relation gen expresses the *generalization* structure for object types. If x is a generalization of y, then this is denoted as $x \,\mathsf{gen}\, y$. y is called a *specifier* of x.

   Generalized object types are usually inhomogeneous, as the specifiers have a different structure, i.e. specifiers do not share instances (this is further elaborated in section 5.2.2). Generalized object types are used to reduce schema redundancy, by relating similar action types of specifiers to the generalization.

   In section 5.4.2.4 the *generalization defining rule* is introduced as a rule for being member of a generalized object type.

9. The relation spec is used to describe *specialization.* If x is a specialization of y, then this is denoted as $x \,\mathsf{spec}\, y$. x is also called a *subtype* of y.

   Contrary to generalization, specialization is a restriction mechanism. Specialized object types have the same structure as their supertype, but will participate in extra action types. In the sequel it will be enforced that each specialization hierarchy has a unique top element (see section 5.2.1.3).

   A subtype thus can be seen as a special grouping of instances of some object type, while a generalization is a grouping of instances of different object types.

   In section 5.4.2.4 the *subtype defining rule* is introduced as a rule for being member of a subtype.

The structure $\mathcal{OAI}$ must satisfy a number of axioms. These are discussed in the next subsections.

5.2.1.1 General rules

The first axiom excludes action types to be part of their own activity:

[OAI-1] *(no cycles)* $\mathsf{Actor}(p) \neq \mathsf{Action}(p)$

Object types that do not participate in any action type can not exist in our philosophy, as these object types lack an initialization (birth) action type.

[OAI-2] *(born objects)* $\exists_{p \in \mathcal{P}} [\mathsf{Actor}(p) = x]$

5.2.1.2 Decomposition

Next we consider module types. The relation compr expresses the decomposition structure of a schema. This relation is a partial order:

[OAI-3] *(irreflexive)* $\neg x \,\mathsf{compr}\, x$

[OAI-4] *(transitive)* $x \,\mathsf{compr}\, y \wedge y \,\mathsf{compr}\, z \Rightarrow x \,\mathsf{compr}\, z$

A module type x is called a main module ($\mathsf{Main}(x)$) if it does not occur in any decomposition ($\neg \exists_y [y \,\mathsf{compr}\, x]$). The existence of a unique main module is postulated by the following axiom:

[OAI-5] *(main module)* $\exists!_x [\mathsf{Main}(x)]$

The following predicate is introduced in order to express that object type x occurs in the same decompositions as object type y:

$$x \,\mathsf{relative}\, y \equiv \forall_z [z \,\mathsf{compr}\, x \Rightarrow z \,\mathsf{compr}\, y]$$

Note that this relation is reflexive and transitive. If not all ancestors of object type x are also ancestor of object type y, then x must be part of a decomposition not containing y:

Lemma 5.1 $\neg x \text{ relative } y \iff \exists_z [z \text{ compr } x \wedge \neg z \text{ compr } y]$

Proof:

Suppose $\neg x$ relative y, then for some object type z we have z compr x but not z compr y. The proof in the other direction is obvious.

Finally, we use x family y as a shorthand for x relative $y \wedge y$ relative x. Note that this relation is an equivalence relation.

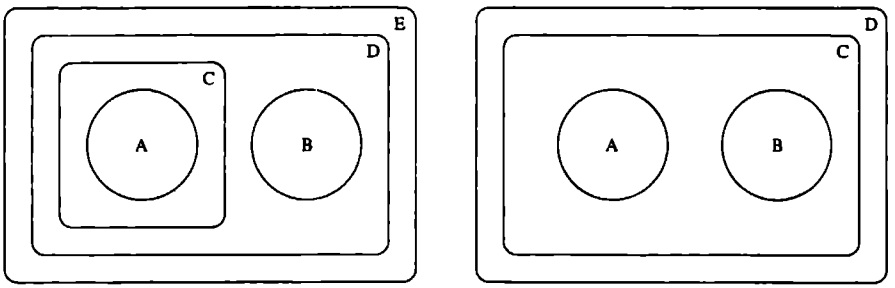


Figure 5.1: Visualization relative and family

The difference between the relations relative and family is visualized in figure 5.1. This figure contains two object action involvement models partially. The left part of the figure describes a schema with object types A and B, and module types C, D and E. In this schema holds B relative A but not A relative B. The schema on the right contains object types A and B, and module types C and D. In this schema A family B holds.

The parent of an object type is the module type which introduces this object type. We introduce the following abbreviation:

$$\text{IsParent}(x, y) \equiv x \text{ compr } y \wedge \neg\exists_z [x \text{ compr } z \wedge z \text{ compr } y]$$

where $\text{IsParent}(x, y)$ expresses that object type y is introduced by module type x. It is required that the module hierarchy is a strict hierarchy:

[OAI-6] *(unique parent)* $\text{IsParent}(x_1, y) \wedge \text{IsParent}(x_2, y) \Rightarrow x_1 = x_2$

We will use $\text{Parent}(x)$ to denote the unique parent of object type x (if $\neg\, \text{Main}(x)$).

Let x be a module type, then the decomposition $\mathcal{OAI}_x$ of x is derived by restricting the basic components of $\mathcal{OAI}$ to the spanning set $\mathcal{O}_x = \{x\} \cup \{y \in \mathcal{O} \mid x \text{ compr } y\}$ of object types[2]. This decomposition should be a valid object action involvement model:

[2]Note that $\mathcal{O}_x = \{x\}$ if x is not a module type

[OAI-7] *(decomposition validity)*

$$x \in \mathcal{M} \Rightarrow \mathcal{OAI}_x \text{ is a valid object action involvement model}$$

A consequence of this rule is, for example, that if an action type is part of a decomposition, then all its participants are also available in this decomposition. In other words, action types can have internal object types as participant, but internal action types can not address themselves to outside object types.

Lemma 5.2 Let m be the main module of the object action involvement model $\mathcal{OAI}$, then $\mathcal{OAI}_m = \mathcal{OAI}$.

Proof:
This is a direct consequence of $\mathcal{O}_m = \mathcal{O}$.

For convenience we introduce $\mathsf{Locals}(m)$ to denote the set of all object types within a module m:

$$\mathsf{Locals}(m) \equiv \left\{ x \;\middle|\; \mathsf{IsParent}(m, x) \right\}$$

5.2.1.3 Subtyping

As specialized object types are groupings of instances of their supertypes, they can have no structure of their own. The same holds for generalized object types:

[OAI-8] *(structural relatedness)* $\mathsf{spec} \cup \mathsf{gen} \subseteq \mathcal{E} \times \mathcal{O}$

Both relations gen and spec are partial orders:

[OAI-9] *(structure* gen*)* gen is a partial order

[OAI-10] *(structure* spec*)* spec is a partial order

As a consequence of structural relatedness, each specialized object type should have a unique *pater familias*, i.e. top. In order to express this, the following predicate is introduced to express y being the pater familias of x:

$$\sqcap(x, y) \equiv \begin{cases} \neg\, \mathsf{IsSpec}(y) & \text{if } x \,\mathsf{spec}\, y \\ x = y & \text{otherwise} \end{cases}$$

where $\mathsf{IsSpec}(y) \equiv \exists_z [y \,\mathsf{spec}\, z]$. Having a unique pater familias is now enforced by:

[OAI-11] *(enforcing pater familias)* $\sqcap(x, y) \wedge \sqcap(x, z) \Rightarrow y = z$

Generalization and specialization relations should not be conflicting, which is the case when a generalized object type is also the specialization of another object type:

[OAI-12] *(gen-spec harmony)* $\mathsf{IsGen}(x) \Rightarrow \neg\,\mathsf{IsSpec}(x)$

where $\mathsf{IsGen}(x) \equiv \exists_y [x \,\mathsf{gen}\, y]$.

Specialized object types can only occur in module types that also contain their supertypes:

[OAI-13] *(spec-compr harmony)* $x \,\mathsf{spec}\, y \Rightarrow y \,\mathsf{relative}\, x$

For generalization an analogous condition is required:

[OAI-14] *(gen-compr harmony)* $x \,\mathsf{gen}\, y \Rightarrow x \,\mathsf{relative}\, y$

The spec-compr harmony is visualized in figure 5.2. Consider the object action involvement model in the upper part of this figure. This schema has a conflicting specialization and module hierarchy. In module type F, object type B is known and object type A is not known. On the other hand B is structurally related to A as A is its supertype. The schema on the bottom shows a specialization and module hierarchy in harmony.

For convenience we introduce Gens and Specs to denote the set of all generalized and specialized object types, respectively:

$$\begin{aligned} \mathsf{Gens} &\equiv \left\{ x \;\middle|\; \mathsf{IsGen}(x) \right\} \\ \mathsf{Specs} &\equiv \left\{ x \;\middle|\; \mathsf{IsSpec}(x) \right\} \end{aligned}$$

5.2.1.4 Example

Figure 5.3 shows a graphical representation of the following algebraic description:

$\mathcal{O}$	$\mathcal{A}$
Main	a_1
A_1	A_2
A_2	a_3
A_3	
A_4	
A_5	
a_1	
a_3	

subtyping
A_4 gen A_1
A_4 gen A_5

$\mathcal{P}$	Action	Actor	χ
p_1	A_2	A_1	r
p_2	A_2	A_3	i
p_3	a_1	A_2	i
p_4	a_1	A_4	r
p_5	a_3	A_3	i
p_6	a_3	A_5	r

The relation compr is obvious, and therefore omitted.

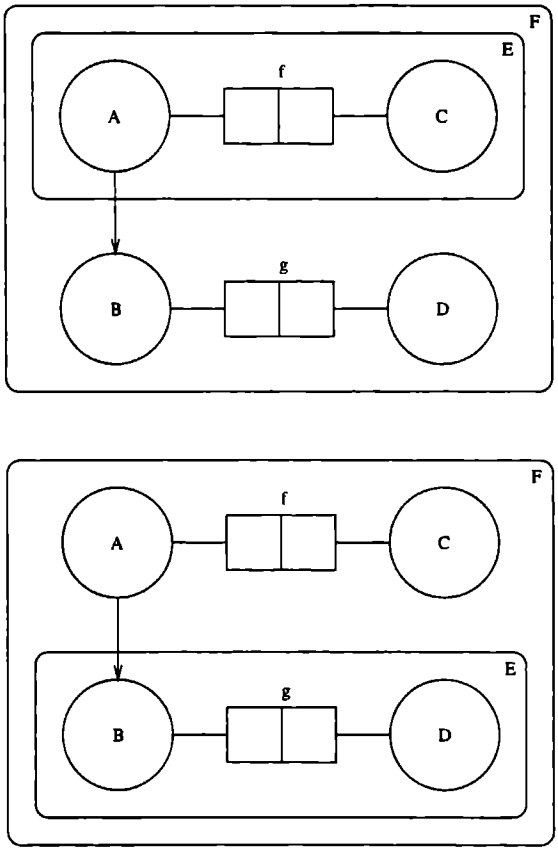


Figure 5.2: Visualization of spec-compr harmony

5.2.2 Type relatedness

A main task of the system analyst is to introduce types for instances occurring in sample sentences. In section 4.2.5 we have described a typing inference mechanism, together with a similarity predicate to enforce typing consistency.

Intuitively, object types can have values in common in some instantiation. For example, object type *Producer* has instances in common with object type *Band* and object type *Person* since object type *Producer* is modeled as a generalization of both *Band* and *Person*. The object action involvement model provides the analyst with concepts, such as generalization, to provide a more subtle qualification of object types. In this section the notion of *type relatedness* is introduced in order to get a grip on such qualifications. Two object types are type related iff this can be proven from the derivation rules which are introduced in the rest of this section. The concept of type relatedness is also useful for query optimization

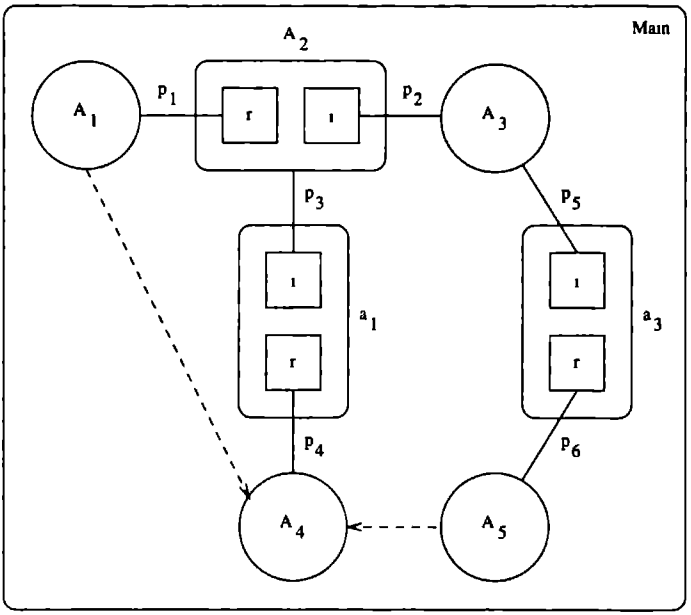


Figure 5.3: A sample object action involvement model

([HPW93]) and the determination of the semantics of constraints ([WHB92]).

Obviously, each object type is type related to itself, and the type relatedness relation is symmetric.

[T-1] *(reflexive)* $\vdash \mathsf{TypeRel}(x, x)$

[T-2] *(symmetric)* $\mathsf{TypeRel}(x, y) \vdash \mathsf{TypeRel}(y, x)$

Object types which can have instances in common with object types occuring in a specialization hierarchy, can have instances in common with all object types of that specialization hierarchy.

[T-3] *(related specialization hierarchy)* $\sqcap(x_1, y) \wedge \sqcap(x_2, y) \wedge \mathsf{TypeRel}(x_1, z) \vdash \mathsf{TypeRel}(x_2, z)$

Since a generalized object type forms a covering object type for all its specifiers, a generalized object type should also cover over object types which are type related with its specifiers.

[T-4] *(related generalization hierarchy)* $x \,\mathsf{gen}\, y \wedge \mathsf{TypeRel}(y, z) \vdash \mathsf{TypeRel}(x, z)$

Group types and sequence type with type related domains (element types) are also type related.

[T-5] *(related groups)* $x, y \in \mathcal{G} \wedge \mathsf{TypeRel}(\mathsf{Elt}(x), \mathsf{Elt}(y)) \vdash \mathsf{TypeRel}(x, y)$

[T-6] *(related sequences)* $x, y \in \mathcal{S} \wedge \mathsf{TypeRel}(\mathsf{Elt}(x), \mathsf{Elt}(y)) \vdash \mathsf{TypeRel}(x, y)$

Module types can be type related if they have the same structure, i.e. the modules consist of the same number of object types, and if their corresponding spanning sets are type related.

[T-7] *(related modules)* $x, y \in \mathcal{M} \wedge \mathsf{TypeRel}(\mathcal{O}_x, \mathcal{O}_y) \vdash \mathsf{TypeRel}(x, y)$

where type relatedness on sets is defined as

$$\mathsf{TypeRel}(X, Y) \equiv \exists_{\text{bijection } \phi: X \to Y} \forall_{x \in X} [\mathsf{TypeRel}(x, \phi(x))]$$

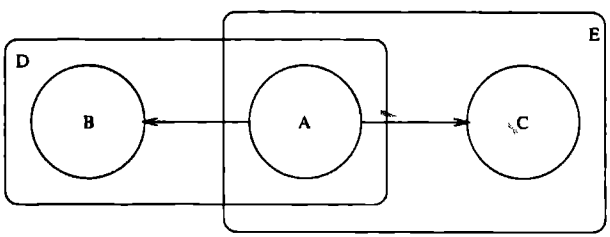


Figure 5.4: Type related modules D and E

Example 5.1
In figure 5.4 module types D and E are type related.

Example 5.2
Consider the object action involvement model of figure 5.3. In this figure the only object types which are (non-reflexively) type related are A_1 and A_4, and, A_5 and A_4.

5.2.3 Inheritance

The concept of type relatedness states whether object types can have instances in common in some instantiation and is used to enforce a consistent typing of the instances of the sample sentences. Once a consistent typing is achieved the concept of *inheritance* can be used to express the existence of common action types and properties for object types.

Intuitively, object types can share action types and properties. For example, object types *Person* and *Band* inherit action type *to produce* from their generalized object type *Producer* (see figure 4.3). An object type x has the potention to inherit action types and properties from an object type y, denoted as $\mathsf{Inherits}(x, y)$, iff this can be proven using the derivation rules introduced in the sequel of this section. The relation Inherits thus describes the *class hierarchy* as introduced in conventional object-oriented analysis methods (see e.g. [CY90]). The concept of inheritance is used to provide a more eloquent paraphrasing mechanism (see section 5.2.7). Section 5.2.6 discusses the semantics of the inheritance relation.

The inheritance relation is a reflexive relation.

[I-1] *(reflexive)* $\vdash \mathsf{Inherits}(x, x)$

Furthermore, object types can inherit action types and properties from their generalized object type and supertypes, respectively.

[I-2] *(specialization inheritance)* $x \,\mathsf{spec}\, y \wedge \mathsf{Inherits}(y, z) \vdash \mathsf{Inherits}(x, z)$

[I-3] *(generalization inheritance)* $y \,\mathsf{gen}\, x \vdash \mathsf{Inherits}(x, y)$

The relation with the concept of type relatedness is expressed by:

Lemma 5.3 $\mathsf{Inherits}(x, y) \Rightarrow \mathsf{TypeRel}(x, y)$

Proof:

Suppose $\mathsf{Inherits}(x, y)$. This proof is by induction on the proof of the inherit relation $\mathsf{TypeRel}(x, y)$. We distinguish two cases:

1. $x = y$. Then we have to prove $\mathsf{TypeRel}(x, x)$. This is true by axiom T-1.
2. $x \neq y$. We distinguish two cases:
   (a) Suppose there exists a z such that $x \,\mathsf{spec}\, z$ and $\mathsf{Inherits}(z, y)$. By induction we know that $\mathsf{Inherits}(z, y)$ implies $\mathsf{TypeRel}(z, y)$. Both x and y have the same pater familias. Applying axiom T-3 leads to $\mathsf{TypeRel}(x, y)$.
   (b) Suppose $y \,\mathsf{gen}\, x$. Via axiom T-1, we know that $\mathsf{TypeRel}(x, x)$. Applying axioms T-2 and T-4 leads to $\mathsf{TypeRel}(x, y)$.

Thus $\mathsf{TypeRel}(x, y)$.

5.2.4 Events of object action involvement model

The semantics of an object action involvement model is the set of its possible logbooks, where the events are conforming to the structure layed down in the object action involvement model. The expression $\mathsf{IsLog}(\mathcal{OAI}, \mathsf{L})$ expresses that L is a logbook of object action involvement model $\mathcal{OAI}$. A logbook L of an object action involvement model is in this approach a mapping:

$$\mathsf{L} : \mathcal{O} \rightarrow \wp^{\mathrm{fin}}(\mathsf{Time} \times \Omega)$$

where Ω includes a set of object identifiers, and Time the set of time stamps. This instantiation mechanism can be described formally via the category **TimeStampSet**, using the category theoretical approach from chapter 7, where it has been shown to be a general population mechanism that enforces all usual properties of populations. For example, the requirement that the time stamp of an action may not precede the time stamp of its involved instances is formulated within this framework.

As an example, the population of action type A_2 (to record), corresponding to the sample population of table B.1, is described as:

$$\{\langle 21\text{-}06\text{-}1967, p_1 : \text{The Rolling Stones}, p_2 : \text{Paint It Black}\rangle, \\ \langle 29\text{-}04\text{-}1991, p_1 : \text{Playful Plebs}, p_2 : \text{I Want You}\rangle\}$$

The labels are used to denote the abstract instances to improve readability (labels are introduced in the object property model, see section 5.3).

5.2.5 Naming and verbalization rules

The object action involvement model is made communicable by naming its elements and providing rules to verbalize its structure. For the naming elements we assume a name space **Names** and a qualification operator $\odot$: **Names** $\times$ **Names** $\rightarrow$ **Names** such that:

$$x_1 \odot y_1 = x_2 \odot y_2 \Rightarrow x_1 = x_2 \wedge y_1 = y_2$$

The action types which are also involved in some predicator, i.e. action types which are objectified, are treated in a special way. First we introduce the following auxiliary predicate:

$$\mathsf{Objectified}(x) \equiv x \in \mathcal{A} \wedge \exists_p [\mathsf{Actor}(p) = x]$$

Object types involved in some predicator are called *active*. The set $\mathcal{I}$ of all active object types is defined by:

$$\mathcal{I} = \mathcal{O} \setminus \mathcal{A} \cup \left\{ x \;\middle|\; \mathsf{Objectified}(x) \right\}$$

In the sequel of this thesis the terms object types and active object types are used interchangeably whenever this will not lead to confusions.

The naming structure consists of the following components:

1. The functions INm, ANm, and PNm assign a name to active object types, action types, and predicators, respectively. The object type names and action type names are used to identify object types and action types within their context. We assume names for object types and action types to be different.

   Predicator names are used to denote how the actor is involved in the action associated with that predicator.

2. The functions IVerbs and AVerbs assign a set of verbalization rules to each active object type and action type, respectively. The verbalizing sets are used for paraphrasing the object action involvement model to natural language sentences. Each verbalization rule corresponds to the structure of some sample sentences from the sample logbook.

The naming structure has to fulfill a number of requirements. A first requirement is that different object types can be denoted differently. As names for object types need not to be unique, unique object denotations are formed by qualification with the module denotations in which they are introduced. For this purpose, the family name $\mathsf{FNm}(x)$ of object type x is introduced as its shortest qualification leading to a unique denotation:

$$\mathsf{FNm}(x) = \begin{cases} \mathsf{INm}(x) & \text{if } \mathsf{INm}(x) \text{ is unique or } \mathsf{Main}(x) \\ \mathsf{FNm}(\mathsf{Parent}(x)) \odot \mathsf{INm}(x) & \text{otherwise} \end{cases}$$

Object types should have unique family names:

[N-1] *(unique family name)* $\mathsf{FNm}(x) = \mathsf{FNm}(y) \Rightarrow x = y$

As a consequence, object types within the same module must have different names:

Lemma 5.4 $\mathsf{INm}(x) = \mathsf{INm}(y) \wedge x \neq y \Rightarrow \neg x \,\mathsf{family}\, y$

Proof:

Suppose $\mathsf{INm}(x) = \mathsf{INm}(y) \wedge x \neq y$. As object types have a unique family name and $x \neq y$ we know that $\mathsf{FNm}(x) \neq \mathsf{FNm}(y)$. Together with the fact that each object type has a unique parent and $\mathsf{INm}(x) = \mathsf{INm}(y)$ it holds that $\mathsf{FNm}(\mathsf{Parent}(x)) \neq \mathsf{FNm}(\mathsf{Parent}(y))$ and thus $\mathsf{Parent}(x) \neq \mathsf{Parent}(y)$. As a consequence $\neg\, \mathsf{Parent}(x) \,\mathsf{compr}\, y$. On the other hand $\mathsf{Parent}(x) \,\mathsf{compr}\, x$. Applying lemma 5.1 leads to $\neg x \,\mathsf{relative}\, y$ and thus $\neg x \,\mathsf{family}\, y$.

Action types may also have the same name, as long as they can be distinguished by the (family) names of their actors:

[N-2] *(action naming)* $\mathsf{ANm}(a) = \mathsf{ANm}(b) \wedge \neg a \,\mathsf{family}\, b \Rightarrow \exists_{p \in a \wedge q \in b} [\mathsf{Actor}(p) \neq \mathsf{Actor}(q)]$

A unique naming for action types thus is obtained by involving the actors. Let action type a consist of predicators $p_1, \ldots, p_k$, such that the family names of its actors are alphabetically ordered. Then an action type is uniquely denoted within its parent module $\mathsf{Parent}(a)$ as:

$$\mathsf{ANm}'(a) = \begin{cases} \mathsf{ANm}(a) & \text{if } \mathsf{ANm}(a) \text{ is unique} \\ \mathsf{ANm}(a) \odot \mathsf{FNm}(\mathsf{Actor}(p_1)) \odot \ldots \odot \mathsf{FNm}(\mathsf{Actor}(p_k)) & \text{otherwise} \end{cases}$$

This is the basis for the family name $\mathsf{FNm}(a)$ for action type a:

$$\mathsf{FNm}(a) = \begin{cases} \mathsf{ANm}'(a) & \text{if } \mathsf{ANm}'(a) \text{ is unique} \\ \mathsf{FNm}(\mathsf{Parent}(a)) \odot \mathsf{ANm}'(a) & \text{otherwise} \end{cases}$$

An analogous requirement is posed for predicator names:

[N-3] *(predicator naming)* $\mathsf{PNm}(p) = \mathsf{PNm}(q) \wedge p \neq q \Rightarrow \mathsf{Action}(p) \neq \mathsf{Action}(q)$

We can uniquely denote each predicator as follows:

$$\mathsf{FNm}(p) = \begin{cases} \mathsf{PNm}(p) & \text{if } \mathsf{PNm}(p) \text{ is unique} \\ \mathsf{FNm}(\mathsf{Action}(p)) \odot \mathsf{PNm}(p) & \text{otherwise} \end{cases}$$

A verbalization rule of a schema element (object type or predicator) is a context-free grammar rule describing how to put it into words. The verbalization of a schema element is a set of associated verbalization rules. The verbalization of a schema element is also referred to as its concrete type. Note that verbalization rules for object types can be enriched for example with synonyms of object type names using lexica such as WordNet ([MBF+93]), see also appendix A.

The first restriction on forming verbalization rules states that object types have different concrete typing:

[V-1] *(unique object verbalization)* $\mathsf{IVerbs}(x) = \mathsf{IVerbs}(y) \Rightarrow x = y$

Action types have extra verbalizations, originating from the sentence structures derived from the sample sentences.

[V-2] *(unique action verbalization)* $\mathsf{AVerbs}(a) = \mathsf{AVerbs}(b) \Rightarrow a = b$

Example 5.3

By naming the elements of figure 5.3, figure 4.3 appears:

$\mathcal{I}$	INm
Main	*Main*
A_1	*Band*
A_2	*Recording*
A_3	*Song*
A_4	*Producer*
A_5	*Person*

$\mathcal{A}$	ANm
a_1	*to produce*
a_2	*to record*
a_3	*to write*

$\mathcal{P}$	PNm
p_1	*agent*
p_2	*object*
p_3	*object*
p_4	*agent*
p_5	*object*
p_6	*agent*

Note that some predicators have the same predicator name. Using family names provides a unique name:

$\mathcal{P}$	FNm
p_1	*to record* ⊙ *agent*
p_2	*to record* ⊙ *object*
p_3	*to produce* ⊙ *object*
p_4	*to produce* ⊙ *agent*
p_5	*to write* ⊙ *object*
p_6	*to write* ⊙ *agent*

The verbalizations of these schema elements are as follows:

$\mathcal{I}$	IVerbs
A_1	"Pop group"
A_2	"the recording of" $\langle p_2 \rangle$ "recorded by" $\langle p_1 \rangle$

$\mathcal{A}$	AVerbs
a_1	$\langle p_4 \rangle$ "produces" $\langle p_3 \rangle$
a_1	$\langle p_3 \rangle$ "is produced by" $\langle p_4 \rangle$
A_2	$\langle p_1 \rangle$ "records" $\langle p_2 \rangle$
A_2	$\langle p_2 \rangle$ "is recorded by" $\langle p_1 \rangle$
a_3	$\langle p_6 \rangle$ "writes" $\langle p_5 \rangle$
a_3	$\langle p_5 \rangle$ "is written by" $\langle p_6 \rangle$

Note that object types can also be verbalized by their object name. This is omitted in the table. However, the table has an entry which provides an alternative verbalization for bands. Furthermore, the table provides a more elaborated object verbalization for recordings.

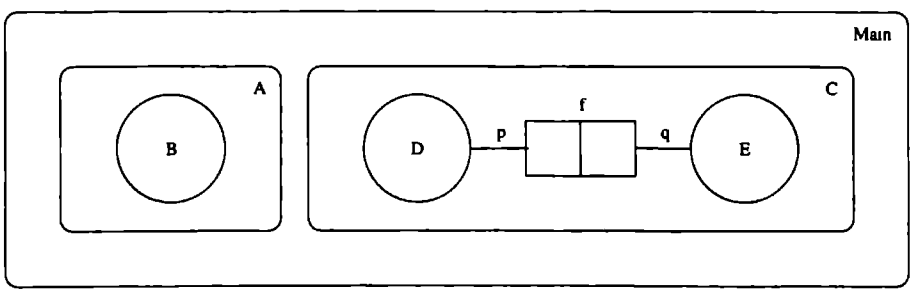


Figure 5.5: A liberal naming mechanism (see example 5.4)

Example 5.4

The naming mechanism as described in this section allows system analysts enough freedom. For figure 5.5 the following family names are allowed:

$\mathcal{O} \cup \mathcal{P}$	FNm
Main	*Main*
A	*Main* $\odot$ *Main*
B	*Main* $\odot$ *Main* $\odot$ *Main*
C	*Main* $\odot$ *Unimportant*
D	*Main* $\odot$ *Unimportant* $\odot$ *Main*
E	*Main* $\odot$ *Unimportant* $\odot$ *Unimportant*
f	*Main* $\odot$ *Unimportant* $\odot$ *Action*
p	*Main* $\odot$ *Unimportant* $\odot$ *Main*
q	*Main* $\odot$ *Unimportant* $\odot$ *Action*

5.2.6 Binding mechanism

In section 5.2.3 we introduced a scheme describing potential inheritance between object types. This is usually referred to as the class hierarchy. In this section we describe what will actually be inherited. The binding relation is required during execution of the information grammar when the meaning of a user sentence is to be investigated by the interpreter of the information grammar.

Each object type in the object action involvement model is involved in a number of action types. However, an action type name may be overloaded, leaving the question what action is meant in the context of a particular object type. Associating a name with an action type, in the context of an object type, is called *binding*. The primitive binding rules are described by the predicate $\mathsf{Has} \subseteq \mathcal{O} \times \mathtt{Names} \times \mathcal{A}$ defined as follows:

[B-1] *(primitive binding)* $\vdash \mathsf{Has}(\mathsf{Actor}(p), \mathsf{ANm}(\mathsf{Action}(p)), \mathsf{Action}(p))$

This primitive binding predicate is extended over the class hierarchy, leading to the predicate Binds, as follows:

[B-2] *(basic binding)* $\mathsf{Has}(x, n, a) \vdash \mathsf{Binds}(x, n, a)$

[B-3] *(inheritance)* $\neg\,\mathsf{Has}(x, n, a) \wedge \mathsf{Inherits}(x, y) \wedge \mathsf{Binds}(y, n, a) \vdash \mathsf{Binds}(x, n, a)$

Next we consider binding at run time. Consider an instance i, then this instance has associated a number of object types at each moment. The predicate $\mathsf{HasType}(\mathsf{L}, i, x)$ states that at the end of logbook L instance i is of type x. The set of all object types associated with instance i after L is referred to as the class hierarchy of i after L. To identify the

meaning of a name n in the context of this instance, we determine whether object type x, associated with i after L, assigns a meaning to name n:

$$\mathsf{IsAwareOf}(\mathsf{L}, i, n, x) \equiv \mathsf{HasType}(\mathsf{L}, i, x) \wedge \exists_a [\mathsf{Binds}(x, n, a)]$$

From the class hierarchy we select the deepest descendants (types) of instance i knowing n in the context of L from this partial family tree:

$$\mathsf{IsDefiner}(\mathsf{L}, i, n, x) \equiv \mathsf{IsAwareOf}(\mathsf{L}, i, n, x) \wedge \forall_y [\mathsf{IsAwareOf}(\mathsf{L}, i, n, y) \wedge \mathsf{Inherits}(y, x) \Rightarrow x = y]$$

If there are more definers, the name n has more than one meaning (*multiple inheritance*). Multiple inheritance can be excluded by the following axiom:

[B-4] *(no multiple inheritance)* $\mathsf{IsDefiner}(\mathsf{L}, i, n, x) \wedge \mathsf{IsDefiner}(\mathsf{L}, i, n, y) \Rightarrow x = y$

In a later section, properties of object types are introduced. The inheritance of properties is derived via the action types which update and inspect these properties.

Remark 5.1

The role of inheritance for conceptual modeling and consequences of multiple inheritance remain a actual research topic. For more readings about (multiple) inheritance the reader is referred to [Bra83], [Car84], [Wil96b], or [SD96b].

5.2.7 Paraphrasing mechanism

Next we introduce a paraphrasing mechanism for the object action involvement model. The paraphrasing mechanism is presented as a context-free grammar, which will be introduced in the sequel of this section. The rules of this grammar are referred to as *paraphrasing rules*. The paraphrasing mechanism should be able to produce a textual description of the structure of all events that may occur in the UoD. Furthermore, a sketch of an algorithm in pseudo code, which collects all grammar rules for the information grammar, is presented. For an example of an experiment using this paraphrasing mechanism, see chapter 6 and appendix D.

5.2.7.1 General format of paraphrasing rules

The verbalization rules for action types, and the names for object and action types form the point of departure for the formulation of the paraphrasing rules. We will generate paraphrasing rules in the style of *affix grammars* (see for example [Kos70]), a family of two-level grammars. For the implementation of the information grammar the AGFL system then can be used. Section 6.2 contains a short introduction to both the formalism and system. The information grammar is demonstrated in section 6.3.

The first level consists of context-free production rules containing affixes. The second level defines the domains of these affixes by meta rules. We will distinguish two types of meta rules: domain *independent* (e.g. singular or plural form affixes), and domain *dependent* (e.g. generalizations). The production rules on the first level are either provided by the domain expert (verbalization rules) or deduced from the algebraic description of the object action involvement model.

5.2.7.2 Meta rules

The domain independent meta rules provide the domains for the affixes ⟨Γ⟩, ⟨number⟩ and ⟨mode⟩:

⟨Γ⟩ :: r | i
⟨number⟩ :: singular | plural
⟨mode⟩ :: actual | structural

The role characterization is enclosed in the first meta rule. The second meta rule provides a mechanism to produce sentences using nouns in plural or singular form. An example of the usage of this meta rule is presented by the introduction of the paraphrasing rules for group types. The last meta rule is used to paraphrase either the actions (events) within a module or the structure of the module. The usage of the last two meta rules will be discussed later.

The following procedure in pseudo code provides the domain independent meta rules:

```
proc Independent Meta Rules () : G_inf rules ;
   return { ⟨Γ⟩ :: r | i, ⟨number⟩ :: singular | plural, ⟨mode⟩ :: actual | structural}
endproc
```

Furthermore, we assume the existence of a meta rule which has as domain the natural numbers[3]. This meta rule provides the paraphrasing mechanism with the possibility to indicate a preferential treatment of the way object types should be paraphrased.

For each predicator a (domain dependent) meta rule is introduced stating all possible actors of that predicator. The set of possible actors for a predicator p, denoted as $\mathsf{Actors}(p)$, is defined as the set of all object types which can be involved in the corresponding action type ($\mathsf{Action}(p)$), i.e.

$$\mathsf{Actors}(p) = \left\{ x \;\middle|\; \mathsf{Binds}(x, \mathsf{ANm}(\mathsf{Action}(p)), \mathsf{Action}(p)) \right\}$$

In the context of figure 4.3 and the example of section 5.2.1.4 predicator p_4, which is contained in action type *to produce*, has as possible actors: *Producer*, *Band*, and *Person*.

For each predicator the following meta rule is introduced:

$$\langle p \rangle :: |_{x \in \mathsf{Actors}(p)}\ \mathsf{FNm}(x)$$

For our example, this results in:

$$\langle p_4 \rangle :: \text{Producer} \mid \text{Band} \mid \text{Person}$$

This leads to the following pseudo code procedure:

```
proc Dependent Meta Rules (M m) : G_inf rules ;
   return {⟨p⟩ :: |_{x∈Actors(p)} FNm(x) | ∃_{a∈Locals(m)∩A} [p ∈ a]}
endproc
```

[3] In the sample information grammar of appendix D the left-side nonterminal for this meta rule is denoted by PREF.

5.2.7.3 Object types

The verbalization rules for object types form the starting point for constructing their paraphrasing rules. For object type x the following paraphrasing rules are introduced:

$$\langle \mathsf{FNm}(x) \rangle \ : \ |_{i=1}^{n(x)} \ \langle \mathsf{FNm}(x)\,[i] \rangle$$
$$\langle \mathsf{FNm}(x)\,[i] \rangle \ : \ v_i$$

where v_i is the i-th verbalization rule of x and $n(x) = |\mathsf{IVerbs}(x)|$ the number of verbalization rules for x. Note that if x has a single verbalization rule v, this can be simplified to:

$$\langle \mathsf{FNm}(x) \rangle \ : \ v$$

The following rules are generated from our running example:

$\langle$Person$\rangle$: "Person"
$\langle$Band$\rangle$: $\langle$Band [1]$\rangle$ | $\langle$Band [2]$\rangle$
$\langle$Band [1]$\rangle$: "Band"
$\langle$Band [2]$\rangle$: "Pop group"

In order to obtain better readable sentences, plural form formulations of object types may be required. Since the name of an object type is assumed to be singular form, the domain expert has to provide the corresponding plural form whenever necessary. In general this leads to the following paraphrasing rules:

$$\langle \mathsf{FNm}(x)\,[\mathsf{Number}] \rangle \ : \ \langle \mathsf{FNm}(x)\,[\mathsf{Singular}] \rangle \ | \ \langle \mathsf{FNm}(x)\,[\mathsf{Plural}] \rangle$$

For example if we consider an object type *Band* to be a group of *Persons* the following paraphrasing rules are added to the information grammar:

$$\langle \mathsf{Person}\,[\mathsf{Number}] \rangle \ : \ \langle \mathsf{Person}\,[\mathsf{Singular}] \rangle \ | \ \langle \mathsf{Person}\,[\mathsf{Plural}] \rangle$$

$$\langle \mathsf{Person}\,[\mathsf{Singular}] \rangle \ : \ \langle \mathsf{Person} \rangle$$
$$\langle \mathsf{Person}\,[\mathsf{Plural}] \rangle \ : \ \text{"Persons"}$$

For each module, the paraphrasing rules for its object types can be collected using the following procedure in pseudo code:

```
proc Paraphrase Module Objects (M m) : G_inf rules ;
   return {⟨FNm(x)⟩ : paraphrasing rule x | x ∈ Locals(m) ∩ I}
endproc
```

5.2.7.4 Action types

Let action type a consist of predicators $p_1, \ldots, p_k$. The paraphrasing rules for action type a are constructed in a number of steps. The first rule is:

$$\langle \mathsf{FNm}(a) \rangle : \begin{cases} \langle \mathsf{FNm}(a)\,[\mathrm{r}] \rangle \mid \langle \mathsf{FNm}(a)\,[\mathrm{i}] \rangle & \text{if } \exists_{p,q \in a}\,[\chi(p) = r \wedge \chi(q) = i] \\ \langle \mathsf{FNm}(a)\,[\mathrm{r}] \rangle & \text{if } \forall_{p \in a}\,[\chi(p) = r] \\ \langle \mathsf{FNm}(a)\,[\mathrm{i}] \rangle & \text{otherwise} \end{cases}$$

This can be further elaborated by extending these rules with a choice of actors for each role in action type a:

$$\langle \mathsf{FNm}(a)\,[\Gamma] \rangle : \langle \mathsf{FNm}(a)\,[\Gamma, \mathsf{FNm}(x_1), \ldots, \mathsf{FNm}(x_k)] \rangle$$

where each $x_i \in \mathsf{Actors}(p_i)$. Each such paraphrasing rule is actualized by employing a verbalization rule from $\mathsf{AVerbs}(a)$. Let $\omega_0 \langle p_{n_1} \rangle \omega_1 \ldots \omega_{l-1} \langle p_{n_l} \rangle \omega_l$ such a verbalization rule, then this leads to the following paraphrasing rule:

$$\langle \mathsf{FNm}(a)\,[\Gamma, \mathsf{FNm}(x_1), \ldots, \mathsf{FNm}(x_k)] \rangle : \omega_0 \langle p_{n_1}\,[\mathsf{FNm}(x_{n_1})] \rangle \omega_1 \ldots \omega_{l-1} \langle p_{n_l}\,[\mathsf{FNm}(x_{n_l})] \rangle \omega_l$$

Next we work out the predicator variants. For predicator p hooked to x the following rule is generated:

$$\langle p\,[\mathsf{FNm}(x)] \rangle : \langle \mathsf{FNm}(x) \rangle$$

Note that the system analyst may decide for a special naming strategy, for example:

$$\langle p\,[\mathsf{FNm}(x)] \rangle : \langle \mathsf{FNm}(x)\,[\mathrm{Plural}] \rangle$$

Example 5.5

As an example of a paraphrasing rule for action types, we consider action type *to produce*.

⟨to produce⟩ : ⟨to produce [r]⟩ | ⟨to produce [i]⟩
⟨to produce [r]⟩ : ⟨to produce [r, Recording, Producer]⟩ |
⟨to produce [r, Recording, Band]⟩ |
⟨to produce [r, Recording, Person]⟩
⟨to produce [i]⟩ : ⟨to produce [i, Recording, Producer]⟩ |
⟨to produce [i, Recording, Band]⟩ |
⟨to produce [i, Recording, Person]⟩

⟨to produce [r, Recording, Producer]⟩ : ⟨p_4 [Producer]⟩ produces ⟨p_3 [Recording]⟩
⟨to produce [r, Recording, Band]⟩ : ⟨p_4 [Band]⟩ produces ⟨p_3 [Recording]⟩
⟨to produce [r, Recording, Person]⟩ : ⟨p_4 [Person]⟩ produces ⟨p_3 [Recording]⟩

⟨to produce [1, Recording, Producer]⟩ : ⟨p_3 [Recording]⟩ is produced by ⟨p_4 [Producer]⟩
⟨to produce [1, Recording, Band]⟩ : ⟨p_3 [Recording]⟩ is produced by ⟨p_4 [Band]⟩
⟨to produce [1, Recording, Person]⟩ : ⟨p_3 [Recording]⟩ is produced by ⟨p_4 [Person]⟩

⟨p_4 [Producer]⟩ : ⟨Producer⟩
⟨p_4 [Band]⟩ : ⟨Band⟩
⟨p_4 [Person]⟩ : ⟨Person⟩
⟨p_3 [Recording]⟩ : ⟨Recording⟩

For each module, the paraphrasing rules for its action types can be collected using the following procedure in pseudo code:

proc Paraphrase Module Actions ($\mathcal{M}$ m) : G_{inf} rules ;
  if Locals(m) $\cap$ $\mathcal{A}$ = $\varnothing$
  then return $\varnothing$
  else return $\{ \langle \mathsf{FNm}(m)\ [\text{actual}] \rangle : +\!\!+_{a \in \mathsf{Locals}(m) \cap \mathcal{A}} \langle a \rangle \}$
    $\cup \{ \langle a \rangle$: paraphrasing rule $a \mid a \in \mathsf{Locals}(m) \cap \mathcal{A} \}$
  endif
endproc

where $+\!\!+$ is the concatenation operator for strings.

5.2.7.5 Paraphrasing structure

The subtyping structure is paraphrased via a rule which describes the generalization structure, and a rule for the specialization structure. Suppose the generalization structure of a module m consists of the generalizations x_1 gen $y_1, \ldots, x_k$ gen y_k. Then this is described by the rule:

$$\langle \text{Gen} \odot \mathsf{FNm}(m) \rangle : \begin{array}{l} \langle \text{Gen} \odot \mathsf{FNm}(x_1) \odot \mathsf{FNm}(y_1) \rangle \\ \vdots \\ \langle \text{Gen} \odot \mathsf{FNm}(x_k) \odot \mathsf{FNm}(y_k) \rangle \end{array}$$

with for each each generalization relation x gen y:

$$\langle \text{Gen} \odot \mathsf{FNm}(x) \odot \mathsf{FNm}(y) \rangle : \langle \mathsf{FNm}(y) \rangle \text{ "is a" } \langle \mathsf{FNm}(x) \rangle \text{ "."}$$

The specialization structure of a module m is handled analogously.

Group types of a module m are paraphrased via a special rule with left hand side non-terminal ⟨Group $\odot$ FNm(m)⟩. Suppose module m has groups $g_1, \ldots, g_k$. Then the groups structures are described in the following way:

$$\langle \text{Group} \odot \text{FNm}(m) \rangle : \langle \text{Group} \odot \text{FNm}(g_1) \rangle \ldots \langle \text{Group} \odot \text{FNm}(g_k) \rangle$$

with for each group type g:

$$\langle \text{Group} \odot \text{FNm}(g) \rangle : \langle \text{FNm}(g) \rangle \text{ "is a group of" } \langle \text{FNm}(\text{Elt}(g))_{\text{[plur]}} \rangle \text{ "."}$$

Paraphrasing of sequence types is almost the same as for group types. Suppose module type m has sequence types $s_1, \ldots, s_k$. This results in the following paraphrasing rule:

$$\langle \text{Seq} \odot \text{FNm}(m) \rangle : \langle \text{Seq} \odot \text{FNm}(s_1) \rangle \ldots \langle \text{Seq} \odot \text{FNm}(s_k) \rangle$$

with for each sequence type s:

$$\langle \text{Seq} \odot \text{FNm}(s) \rangle : \langle \text{FNm}(s) \rangle \text{ "is a sequence of" } \langle \text{FNm}(\text{Elt}(s))_{\text{[plur]}} \rangle \text{ "."}$$

The last decomposable object types to consider are the module types. Suppose module type m consist of object types $x_1, \ldots x_n$, such that $x_i \in \text{Locals}(m)$. The paraphrasing rule for module type m then is:

$$\begin{aligned} \langle \text{Mod} \odot \text{FNm}(m) \rangle : \; & \langle \text{FNm}(m) \rangle \text{ "is a composition of"} \\ & \text{FNm}(x_1) \text{","} \ldots \text{","} \; \text{FNm}(x_n) \text{"."} \end{aligned}$$

For each module, the paraphrasing rules for its structure can be collected using the following procedures in pseudo code:

```
proc Paraphrase Module Structure (M m) : G_inf rules ;
   return {⟨FNm(m) [structure]⟩ : ⟨Gen ⊙ FNm(m)⟩
                                  ⟨Spec ⊙ FNm(m)⟩
                                  ⟨Group ⊙ FNm(m)⟩
                                  ⟨Seq ⊙ FNm(m)⟩
                                  ⟨Mod ⊙ FNm(m)⟩}
        ∪ Paraphrase Module Generalization(m)
        ∪ Paraphrase Module Specialization(m)
        ∪ Paraphrase Module Groups(m)
        ∪ Paraphrase Module Sequences(m)
        ∪ Paraphrase Module Decomposition(m)
endproc

proc Paraphrase Module Generalization (M m) : G_inf rules ;
   return { ⟨Gen ⊙ FNm(m)⟩ : +_{x∈Locals(m)∩Gens} ⟨Gen ⊙ FNm(x) ⊙ FNm(y)⟩ }
        ∪ {⟨Gen ⊙ FNm(x) ⊙ FNm(y)⟩ : ⟨FNm(y)⟩ "is a" ⟨FNm(x)⟩ "." | x gen y }
endproc

proc Paraphrase Module Specialization (M m) : G_inf rules ;
   return { ⟨Spec ⊙ FNm(m)⟩ : +_{x∈Locals(m)∩Specs} ⟨Spec ⊙ FNm(x) ⊙ FNm(y)⟩ }
        ∪ {⟨Spec ⊙ FNm(x) ⊙ FNm(y)⟩ : ⟨FNm(x)⟩ "is a" ⟨FNm(y)⟩ "." | x spec y }
endproc
```

proc Paraphrase Module Groups ($\mathcal{M}$ m) : G_{inf} rules ;
  return $\{$ ⟨Group ⊙ FNm(m)⟩ : $+\!\!+_{g \in \text{Locals}(m) \cap \mathcal{G}}$ ⟨Group ⊙ FNm(g)⟩ $\}$
    ∪{⟨Group ⊙ FNm(x)⟩ : ⟨FNm(g)⟩ "is a group of" ⟨FNm(Elt(g)) [plur]⟩ "."
      | $g \in$ Locals(m) ∩ $\mathcal{G}$}
endproc

proc Paraphrase Module Sequences ($\mathcal{M}$ m) : G_{inf} rules ;
  return $\{$ ⟨Seq ⊙ FNm(m)⟩ : $+\!\!+_{s \in \text{Locals}(m) \cap \mathcal{S}}$ ⟨Seq ⊙ FNm(s)⟩ $\}$
    ∪{⟨Seq ⊙ FNm(x)⟩ : ⟨FNm(s)⟩ "is a sequence of" ⟨FNm(Elt(s)) [plur]⟩ "."
      | $s \in$ Locals(m) ∩ $\mathcal{S}$}
endproc

proc Paraphrase Module Decomposition ($\mathcal{M}$ m) : G_{inf} rules ;
  return { ⟨Mod ⊙ FNm(m)⟩ : ⟨FNm(m)⟩ "is a composition of" $+\!\!+_{x \in \text{Locals}(m)}$ FNm(x)"." }
endproc

5.2.7.6 Driver

In the previous sections procedures have been introduced which collect domain independent meta rules, domain dependent rules, paraphrasing rules for object types, paraphrasing rules for action types, and rules for the structure of a module. In this section an elaboration (a driver) on these procedures is described.

The driver is described in such a way that it is possible to focus on two aspects of the information grammar:

1. the structure, object types, and action types of a particular module,
2. the structure, object types, and action types of a particular module and all its underlying modules.

The procedure 'Paraphrase Module' returns information grammar (G_{inf}) rules for describing the structure of a particular module.

proc Paraphrase Module ($\mathcal{M}$ m) : G_{inf} rules ;
  return
    {⟨FNm(m)⟩ : ⟨FNm(m) [mode]⟩} ∪
    {⟨FNm(m) [mode]⟩ : ⟨FNm(m) [actual]⟩ | ⟨FNm(m) [structural]⟩} ∪
    Paraphrase Module Objects(m) ∪
    Paraphrase Module Actions(m) ∪
    Paraphrase Module Structure(m) ∪
    Independent Meta Rules ∪
    Dependent Meta Rules(m)
endproc

For obtaining information grammar rules of particular module and all its underlying modules the driver is extended with:

proc Paraphrase Modules ($\mathcal{M}$ m) : $\mathsf{G}_{\mathsf{inf}}$ rules ;
  return $\bigcup_{x \in \mathcal{M}_m}$ Paraphrase Module (x)
endproc

For obtaining the complete information grammar of the object action involvement model the previous procedure, 'Paraphrase Modules', can be reused as the object action involvement model is also a module (*Main*). Thus this results in the following procedure:

proc Paraphrase $\mathcal{OAI}$ () : $\mathsf{G}_{\mathsf{inf}}$ rules ;
  return Paraphrase Modules (Main)
endproc

The information grammar is constructed in such a way that it is possible to focus on particular aspects of the object action involvement model. For example, the complete structure of the object action involvement model can be obtained from the start symbol $\langle \text{Main}_{\text{[structural]}} \rangle$.

5.3 Object property model

In this section formal aspects (syntax and semantics) of the object property model are discussed. An algorithm to obtain a part of the information grammar (that part which is described by the object property model) is described. Furthermore, the integration with the object action involvement model is elaborated.

5.3.1 Syntax

In the object action involvement model only abstract object types are considered. In this section the set of object types $\mathcal{O}$ is extended with a set $\mathcal{L}$ of concrete object types, the so-called *label types*, i.e. $\mathcal{L} \subseteq \mathcal{O}$. They can, in contrast with abstract object types, be represented on a communication medium. The extension of the set of object types with the set of label types needs a number of refinements on the formalization of the object action involvement model, if a strict separation between abstract object types and concrete object types is required:

1. Label types are not involved in a subtyping hierarchy, i.e. $\mathsf{spec} \cup \mathsf{gen} \subseteq \mathcal{E} \times \mathcal{O} \setminus \mathcal{L}$.
2. The strict separation between abstract and concrete object types prohibits label types to act as an element type, i.e. $\mathsf{Elt}(x) \notin \mathcal{L}$.

Formally, the *object property model* is a structure $\mathcal{OP}$ consisting of the following basic aspects:

1. A set $\mathcal{B}$ of properties. A property *bridges* the gap between object types and label types. With each property p a *retrieval* and *update* action type is associated ($\mathsf{Retrieve}(p)$ and $\mathsf{Update}(p)$ respectively).

2. A relation Triggers between action types and update action types for properties. The expression $\mathsf{Triggers}(a, \mathsf{Update}(p))$ states that action type a triggers action type $\mathsf{Update}(p)$.

3. A relation IsDen between object types and properties. The expression $\mathsf{IsDen}(x, p)$ states that property p should be used to denote instances of object type x.

Action type a is binary if:

$$\mathsf{Binary}(a) \equiv \exists_{p,q}\,[\mathsf{Action}(p) = a \wedge \mathsf{Action}(q) = a \wedge \forall_r\,[\mathsf{Action}(r) = a \Rightarrow r = p \vee r = q]]$$

Retrieval and update operations for properties are binary action types.

[OP-1] *(binary bridges)* $\mathsf{Binary}(\mathsf{Retrieve}(p)) \wedge \mathsf{Binary}(\mathsf{Update}(p))$

We will also use the abbreviation $r \in \mathsf{Bridge}(p)$ for $\mathsf{Action}(r) = \mathsf{Retrieve}(p) \vee \mathsf{Action}(r) = \mathsf{Update}(p)$. Furthermore we introduce the predicates:

$$\mathsf{Object}(p) = x \equiv x \notin \mathcal{L} \wedge \exists_{r \in \mathsf{Bridge}(p)}\,[\mathsf{Actor}(r) = x]$$
$$\mathsf{Label}(p) = x \equiv x \in \mathcal{L} \wedge \exists_{r \in \mathsf{Bridge}(p)}\,[\mathsf{Actor}(r) = x]$$

Using these predicates the nature of bridges, i.e. the strict separation between abstract and concrete object types, is expressed by:

[OP-2] *(unique property owner)* $\exists!_x\,[\mathsf{Object}(p) = x]$

[OP-3] *(unique property domain)* $\exists!_x\,[\mathsf{Label}(p) = x]$

Label types can only play a role in action types that bridge the gap between abstract and concrete object types:

[OP-4] *(label involvement)* $\mathsf{Actor}(r) \in \mathcal{L} \Rightarrow \exists_p\,[r \in \mathsf{Bridge}(p)]$

In such action types, the role of the label type is a passive one:

[OP-5] *(passive label involvement)* $\mathsf{Actor}(r) \in \mathcal{L} \Rightarrow \chi(r) = i$

Each property must be initialized. This requires an action type which triggers the associated update action type.

[OP-6] *(property initialization)* $\exists_a\,[\mathsf{Triggers}(a, \mathsf{Update}(p))]$

Example 5.6

Figure 5.6 shows a graphical representation which includes the following algebraic description (with respect to the object property model):

$\mathcal{L}$	$\mathcal{B}$	Retrieve	Update	Triggers
L	p	$\mathsf{Retrieve}(p)$	$\mathsf{Update}(p)$	$\mathsf{Triggers}(a_4, \mathsf{Update}(p))$

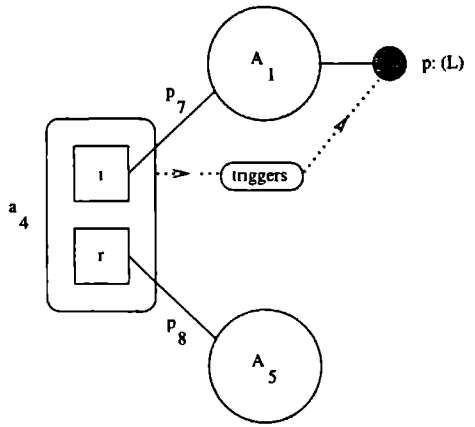


Figure 5.6: Object property model

5.3.2 Populating object property model

The property model is an extension to the object action involvement model. It provides more structure for the way that label types are involved in this model. In order to populate label types the set Ω of object identifiers is extended with concrete values (labels). As a result, the population mechanism of the object action involvement model is still valid.

5.3.3 Naming and verbalization rules

The naming and verbalization functions (INm, ANm, IVerbs, AVerbs) for object types and action types of the object action involvement model did not assume label types, retrieval action types, and update action types to have a name nor verbalization rules. In this section, names and verbalization rules for label types will be assumed (and thus INm and IVerbs can be applied to label types), and standard names and verbalization rules for retrieval and update action types are introduced.

Each property p is also assigned a name and set of verbalization rules using $\mathsf{BNm}(p)$ and $\mathsf{BVerbs}(p)$, respectively. Properties may have the same name name as long as they can be distinguished by the family names of the owners of the property:

[N-4] *(property naming)* $\mathsf{BNm}(p) = \mathsf{BNm}(q) \wedge p \neq q \Rightarrow \mathsf{Object}(p) \neq \mathsf{Object}(q)$

We can uniquely denote each property as follows:

$$\mathsf{FNm}(p) = \begin{cases} \mathsf{BNm}(p) & \text{if } \mathsf{BNm}(p) \text{ is unique} \\ \mathsf{FNm}(\mathsf{Object}(p)) \odot \mathsf{BNm}(p) & \text{otherwise} \end{cases}$$

Example 5.7

In the context of example 5.6, $\mathsf{BNm}(p)$ can be chosen *Band name* whereas $\mathsf{BVerbs}(p)$ can be the set of verbalization rules $\{$"Band name", "Pop group name"$\}$.

Names and verbalization rules for retrieval action types and update action types are generated as follows. Let p be a property with $\mathsf{Retrieve}(p) = \{r_1, r_2\}$ and $\mathsf{Update}(p) = \{u_1, u_2\}$, such that $\mathsf{Actor}(r_1)$ and $\mathsf{Actor}(s_1)$ are (abstract) object types. This leads to:

$$\begin{aligned}
\mathsf{ANm}(\mathsf{Retrieve}(p)) &= \textit{to ask} \odot \mathsf{FNm}(p) \\
\mathsf{ANm}(\mathsf{Update}(p)) &= \textit{to get} \odot \mathsf{FNm}(p) \\
\mathsf{AVerbs}(\mathsf{Retrieve}(p)) &= \{\langle r_1\rangle \text{ "asks" } \langle \mathsf{FNm}(p)\rangle, \\
&\quad \langle \mathsf{FNm}(p)\rangle \text{ "belongs to" } \langle r_1\rangle\} \\
\mathsf{AVerbs}(\mathsf{Update}(p)) &= \{\langle u_1\rangle \text{ "gets" } \langle \mathsf{FNm}(p)\rangle, \\
&\quad \langle \mathsf{FNm}(p)\rangle \text{ "is given to" } \langle u_1\rangle\}
\end{aligned}$$

5.3.4 Inheritance of properties

As stated in section 5.2.3 object types can inherit besides action types also properties. In Section 5.2.6 the semantics of the inheritance relation with respect the inheritance of action types has been discussed. The semantics of the inheritance of properties is analogous and is based on the observation that properties can be modeled via their retrieval and update action types.

The definitions of Has and Binds have to be defined on properties in order to apply the axioms for binding. Object type x has a property p with name $\mathsf{BNm}(p)$, denoted as $\mathsf{Has}(x, n, p)$, where:

$$\mathsf{Has}(x, n, p) \equiv \mathsf{Has}(x, \mathsf{ANm}(\mathsf{Update}(p)), \mathsf{Update}(p)) \wedge \mathsf{BNm}(p) = n$$

Object type x binds name n to property p, denoted as $\mathsf{Binds}(x, n, p)$, where:

$$\mathsf{Binds}(x, n, p) \equiv \mathsf{Binds}(x, \mathsf{ANm}(\mathsf{Update}(p)), \mathsf{Update}(p)) \wedge \mathsf{BNm}(p) = n$$

Lemma 5.5 $x \,\mathsf{spec}\, y \vee y \,\mathsf{gen}\, x \Rightarrow \forall_n [\exists_p [\mathsf{Binds}(y, n, p)] \Rightarrow \exists_q [\mathsf{Binds}(x, n, q)]]$

Proof:

This lemma is only proven for the case $x \,\mathsf{spec}\, y$. The proof for $y \,\mathsf{gen}\, x$ is analogous.

Suppose $x \,\mathsf{spec}\, y$. Furthermore, suppose object type y binds name n to property p. Then we have to prove that object type x binds the same name to a property q. We distinguish two cases:

1. Suppose $\mathsf{Has}(x, n, q)$. As a result of axiom B-2 we know that $\mathsf{Binds}(x, n, q)$.
2. Suppose $\neg\, \mathsf{Has}(x, n, q)$. $x \,\mathsf{spec}\, y$ and axiom I-2 leads to $\mathsf{Inherits}(x, y)$. Via axiom B-3 we can conclude $\mathsf{Binds}(x, n, q)$.

The binding relation for properties states what property names can be used in the context of an object type, and the properties corresponding to such names. The *state record* of an object type is introduced as the set of all properties which can be bind by that object type:

$$\mathsf{StatRec}(x) = \left\{ p \in \mathcal{B} \;\middle|\; \exists_n \left[\mathsf{Binds}(x, n, p) \right] \right\}$$

Next we consider binding of properties at run time. The predicates IsAwareOf and IsDefiner are reused. The expression $\mathsf{IsAwareOf}(\mathsf{L}, i, \mathsf{ANm}(\mathsf{Update}(p)), x)$ answers the question whether instance i, in the context of object type x, after logbook L is aware of property p. The predicate $\mathsf{IsDefiner}(\mathsf{L}, i, \mathsf{ANm}(\mathsf{Update}(p)), x)$ states whether object type x, which is a proper type of instance i after logbook L, has introduced this property p.

5.3.5 Extended paraphrasing mechanism

In this section the paraphrasing mechanism of the object action involvement model is extended with rules for paraphrasing properties. Furthermore, a way to paraphrase instances will be sketched at the end of this section.

5.3.5.1 Meta rules

The domain independent affix rule ⟨mode⟩ is extended with trigger for paraphrasing trigger relations, i.e.

⟨mode⟩ :: actual | structural | trigger

Its usage will be discussed later.

In order to paraphrase properties another domain dependent meta rule is introduced which summarizes all properties within a module m:

$$\langle \text{All props} \odot \mathsf{FNm}(m) \rangle :: |_{p \in \mathcal{B}_m} \mathsf{FNm}(p)$$

where $\mathcal{B}_m$ is the set of all properties which belong to object types within a module m, i.e.

$$\mathcal{B}_m = \left\{ p \;\middle|\; \mathsf{Retrieve}(p) \in \mathsf{Locals}(m) \wedge \mathsf{Update}(p) \in \mathsf{Locals}(m) \right\}$$

As a result the procedure 'Dependent Meta Rules is modified as follows:

proc Dependent Meta Rules ($\mathcal{M}$ m) : $\mathsf{G_{inf}}$ rules ;
  return $\{ \langle p \rangle :: |_{A \in \mathsf{Actors}(p)} \mathsf{FNm}(A) \mid p \in f \cap \mathsf{Locals}(m) \cap \mathcal{A} \}$
    $\cup \{ \langle \text{All props} \odot \mathsf{FNm}(m) \rangle :: |_{p \in \mathcal{B}_m} \mathsf{FNm}(p) \}$
endproc

Example 5.8

Suppose object type *Recording* of figure 4.3 has properties *Studio number* and *Tape number*. This leads to the following meta rule (where the property of object type *Band* is also included):

$\langle \text{All props} \odot \mathsf{FNm}(m) \rangle$ :: Band name | Studio number | Tape number

5.3.5.2 Property triggers

As stated before, properties can be modeled using action types. For action types a mechanism to achieve their paraphrasing rules is already presented in section 5.2.7.4. This mechanism can be applied for the retrieval and update action types of property p, resulting in rules for $\langle \mathsf{FNm}(\mathsf{Update}(p)) \rangle$ and $\langle \mathsf{FNm}(\mathsf{Retrieve}(p)) \rangle$.

However, the resulting set of paraphrasing rules is not sufficient for paraphrasing triggers. Since triggers form a relevant part of the object property model the set of paraphrasing rules for action types is extended with rules for paraphrasing triggers. Let action type a trigger the update action type b for property p, i.e. $\mathsf{Triggers}(a, b)$. This leads to the following paraphrasing rule:

$\langle \mathsf{FNm}(b)\,[\mathsf{FNm}(p)] \rangle$: "when" $\langle \mathsf{FNm}(a) \rangle$ "then" $\langle \mathsf{FNm}(b) \rangle$

which brings us back to the paraphrasing rules for action types presented in section 5.2.7.4.

The paraphrasing rules for triggers are obtained by:

proc Paraphrase Module Triggers ($\mathcal{M}$ m) : $\mathsf{G}_{\mathsf{inf}}$ rules ;
  return $\{\langle \mathsf{FNm}(m)\,[\text{trigger}] \rangle : +\!+_{p \in \mathcal{B}_m} \langle \mathsf{FNm}(\mathsf{Update}(p))\,[\mathsf{FNm}(p)] \rangle\}$
    $\cup \{\langle \mathsf{FNm}(\mathsf{Update}(p))\,[\mathsf{FNm}(p)] \rangle : +\!+_{\mathsf{Triggers}(a,\mathsf{Update}(p))}$ "when" $\langle \mathsf{FNm}(a) \rangle$ "then" $\langle \mathsf{FNm}(\mathsf{Update}(p)) \rangle\}$
endproc

Example 5.9

The paraphrasing rule for the trigger $\mathsf{Triggers}(a_4, \mathsf{Update}(p))$ in the context of figures 4.5 and 5.6 is:

$\langle$to get [Band name]$\rangle$: "when" $\langle$to set up$\rangle$ "then" $\langle$to get $\odot$ Band name$\rangle$

5.3.5.3 Paraphrasing structure

Suppose module m has object types $x_1, \ldots, x_k$. Then the properties of these object types are described by the rule:

$$\langle \text{Props} \odot \mathsf{FNm}(m) \rangle : \langle \text{Props} \odot \mathsf{FNm}(x_1) \rangle \ldots \langle \text{Props} \odot \mathsf{FNm}(x_k) \rangle$$

The properties which belong to object type x of module m are paraphrased via a special rule. Let $\mathsf{StatRec}(x) = \{p_1, \ldots, p_l\}$. Then this is described by the rule:

$$\begin{aligned} \langle \text{Props} \odot \mathsf{FNm}(x) \rangle : \; & \langle \mathsf{FNm}(x) \rangle \text{ "has" } \langle \mathsf{FNm}(p_1) \rangle \text{ "denoted as" } \langle \mathsf{FNm}(\mathsf{Label}(p_1)) \rangle \text{ "."} \\ & \vdots \\ & \langle \mathsf{FNm}(x) \rangle \text{ "has" } \langle \mathsf{FNm}(p_l) \rangle \text{ "denoted as" } \langle \mathsf{FNm}(\mathsf{Label}(p_l)) \rangle \text{ "."} \end{aligned}$$

In order to paraphrase the complete structure (including rules for properties) of a module, the procedure 'Paraphrase Module Structure' is extended with 'Paraphrase Module Properties(m)', where the procedure 'Paraphrase Module Properties' is:

proc Paraphrase Module Properties ($\mathcal{M}$ m) : $\mathsf{G}_{\mathsf{inf}}$ rules ;
  return $\Big\{ \langle \text{Props} \odot \mathsf{FNm}(m) \rangle : \mathbin{+\!\!+}_{x \in \mathsf{Locals}(m) \setminus \mathcal{L}} \langle \text{Props} \odot \mathsf{FNm}(x) \rangle \;\Big|\; \mathsf{StatRec}(x) \neq \varnothing \Big\}$
    $\cup \{ \langle \text{Props} \odot \mathsf{FNm}(x) \rangle : \langle \mathsf{FNm}(x) \rangle$ "has" $\langle \mathsf{FNm}(p) \rangle$ "denoted as" $\langle \mathsf{FNm}(\mathsf{Label}(p)) \rangle$ "."
      $\mid x \in \mathsf{Locals}(m) \setminus \mathcal{L} \wedge p \in \mathsf{StatRec}(x) \}$
endproc

5.3.5.4 Extended driver

The information grammar is extended by paraphrasing rules for triggers. As a result the procedure 'Paraphrase Module' has to be modified slightly by adding 'Paraphrase Module Triggers(m)'. Furthermore, the start rule for paraphrasing a module:

$$\Big\{ \langle \mathsf{FNm}(m)_{\,[\text{mode}]} \rangle : \langle \mathsf{FNm}(m)_{\,[\text{actual}]} \rangle \mid \langle \mathsf{FNm}(m)_{\,[\text{structural}]} \rangle \Big\}$$

is transformed into:

$$\Big\{ \langle \mathsf{FNm}(m)_{\,[\text{mode}]} \rangle : \langle \mathsf{FNm}(m)_{\,[\text{actual}]} \rangle \mid \langle \mathsf{FNm}(m)_{\,[\text{structural}]} \rangle \mid \langle \mathsf{FNm}(m)_{\,[\text{trigger}]} \rangle \Big\}$$

Executing nonterminal $\langle \text{Main}_{\,[\text{trigger}]} \rangle$ leads to a textual description of all trigger relations in the UoD.

5.3.5.5 Instances

Properties are used to denote instances of object types. Using these instance denotations for the paraphrasing mechanism leads to instantiated sentences. Usually, not all properties will be required for instance denotation. The relation IsDen states whether a property is used to denote its instances.

[OP-7] *(denotational correctness)* $\mathsf{IsDen}(x,p) \Rightarrow \exists_y [\mathsf{Inherits}(x,y) \wedge \mathsf{Object}(p) = y]$

We introduce $\mathsf{Den}(x)$ as the set of properties used to denote instances of object type x as follows:

$$\mathsf{Den}(x) = \left\{ p \mid \mathsf{IsDen}(x,p) \right\}$$

No axioms will be added to enforce a unique denotation for each object type. However, some denotation must be possible:

[OP-8] *(denotable objects)* $\mathsf{Den}(x) \neq \varnothing$

As a consequence an instance of object type x is only denotable if it has a non-empty set of properties, i.e. $\mathsf{StatRec}(x) \neq \varnothing$.

The paraphrasing mechanism is equipped with one more domain independent meta rule:

$\langle \mathsf{sort} \rangle$ :: typed | instantiated

This affix rule is used to switch between sentences on a type level and sentences on an instance level. The sentence *Band records Song* is on a type level while the sentence *The Rolling Stones record Paint It Black* is an instantiated sentence.

The meta rule requires a number of extensions of the paraphrasing rules for abstract object types and action types. For each abstract object type x with verbalization rule v, its rule:

$\langle \mathsf{FNm}(x) \rangle$: v

is changed in:

$\langle \mathsf{FNm}(x)_{\text{[typed]}} \rangle$: v
$\langle \mathsf{FNm}(x)_{\text{[instantiated]}} \rangle$: $|_{y \in \mathsf{L}(x)} \langle x : y \rangle$

where $\mathsf{L}(x)$ is a logbook of abstract object type x (see section 5.2.4). An instance of an object type is a tuple consisting of a time stamp, and an object identifier (for abstract object types) or a label (for concrete object types). The projection functions π_t and π_i are used to extract the time component of an instance, and its object identifier or label, respectively. For readability, these operators are often omitted in the sequel.

Let $\mathsf{Den}(x) = \{p_1, \ldots, p_k\}$. Then for each instance y from $\mathsf{L}(x)$ the following rule is generated:

$\langle x : y \rangle$: $\langle \mathsf{FNm}(x)_{\text{[typed]}} \rangle$ "$(\mathsf{Denote}(y,p_1), \ldots, \mathsf{Denote}(y,p_k))$"

where $\mathsf{Denote}(y, p_t)$ is the value of property p_t of instance y, i.e. $\mathsf{Denote}(y, p_t) = \pi_{\mathsf{i}}(l_t)$ with l_t a *valid* and *updated* denotation of property p_t for y. This valid and updated denotation is also referred to as the *current* denotation, $\mathsf{CurDen}(y, p, l)$ where

$$\mathsf{CurDen}(y, p, l) \equiv \begin{cases} y \in \mathsf{L}(\mathsf{Object}(p)) \wedge l \in \mathsf{L}(\mathsf{Label}(p)) \\ \wedge\ \exists_{v \in \mathsf{L}(\mathsf{Update}(p))} \left[\pi_{\mathsf{i}}(v) = \langle y, l \rangle \wedge \forall_{w \in \mathsf{Update}(p)} \left[\pi_{\mathsf{t}}(w) \leq \pi_{\mathsf{t}}(v)\right]\right] \end{cases}$$

The paraphrasing rules for action types are also modified. The affixes typed and instantiated are passed through all paraphrasing rules in a same way as the affixes for role characterization, i.e. the affixes r and i. Finally, the predicator variants, $(\langle p\,[\mathsf{FNm}(x)]\rangle)$ with $x \in \mathsf{Actors}(p)$, are also extended with the affixes typed and instantiated. As a result their paraphrasing rules are changed into:

$\langle p\,[\mathsf{FNm}(x),\text{typed}]\rangle$: $\langle \mathsf{FNm}(x)\,[\text{typed}]\rangle$
$\langle p\,[\mathsf{FNm}(x),\text{instantiated}]\rangle$: $\langle \mathsf{FNm}(x)\,[\text{instantiated}]\rangle$

Example 5.10
Suppose $y \in \mathsf{L}(x)$ with $\mathsf{Den}(x) = \{p\}$. Let $\mathsf{FNm}(x) = \textit{Band}$, $\mathsf{FNm}(p) = \textit{Band name}$, and $\mathsf{Denote}(y, p) =$ The Rolling Stones. This leads to the following revision and extension of the information grammar:

$\langle \text{Band}\,[\text{typed}]\rangle$: $\langle \text{Band}\rangle$
$\langle \text{Band}\,[\text{instantiated}]\rangle$: $\langle x : y\rangle$

$\langle x : y\rangle$: $\langle \text{Band}\,[\text{typed}]\rangle$ "(The Rolling Stones)"

$\langle p_4\,[\text{Band},\text{typed}]\rangle$: $\langle \text{Band}\,[\text{typed}]\rangle$
$\langle p_4\,[\text{Band},\text{instantiated}]\rangle$: $\langle \text{Band}\,[\text{instantiated}]\rangle$

5.4 Object life model

In this section formal aspects (syntax and semantics) of the object life model are discussed. An algorithm to obtain a part of the information grammar (that part which is described by the object life model) is described. Furthermore, the integration with the object action involvement model and object property model is elaborated.

5.4.1 Syntax

The *object life model* extends the object action involvement model and the object property model with the notion of *generalized predicator*. The model adds the following aspects:

1. The set $\mathcal{C}$ of *generalized predicators* consists of the set $\mathcal{P}$ of predicators, and the following sets:

   (a) A set $\mathcal{D}$ of *surrogates* or *deputies*

   (b) A set $\mathcal{X}$ of *repetition predicators.*

   (c) A set $\mathcal{Y}$ of *choice predicators.*

   (d) A set $\mathcal{Z}$ of *merge predicators.*

   In the context of the object life model, predicators and surrogates are also referred to as *tasks* or *task predicators.* The set of tasks is denoted with $\mathcal{T}$. Generalized predicators are also referred to as *components.* Repetition predicators, choice predicators and merge predicators are also called *structural components* or *structural predicators.*

2. The function $\mathsf{Pred} : \mathcal{D} \to \mathcal{P}$ assigns to each surrogate the predicator from which the surrogate is a derative.

3. The partial function $\mathsf{Succ} : \mathcal{C} \mapsto \mathcal{C}$ describes the *sequential* order of components. Component $\mathsf{Succ}(c)$ is called the *successor* of component c.

4. The partial function $\mathsf{Decomp} : \mathcal{C} \mapsto \mathcal{C} \setminus \mathcal{T}$ provides initial components of structural components. The expression $\mathsf{Decomp}(c) = d$ states component c to be a *decomposition alternative* of structural component d.

5. The relation $\mathsf{Connect} \subseteq \mathcal{C} \times \mathcal{C}$ is the irreflexive transitive closure of both sequential order and decomposition.

6. The relation $\mathsf{HasInit} \subseteq \mathcal{O} \times \mathcal{C} \setminus \mathcal{Z}$ describes which components initialize object types.

The relation Connect forms a partial order of components.

[OL-1] *(no cycles)* $c\ \mathsf{Connect}\ d \Rightarrow \neg d\ \mathsf{Connect}\ c$

[OL-2] *(transitivity)* $c\ \mathsf{Connect}\ d \wedge d\ \mathsf{Connect}\ e \Rightarrow c\ \mathsf{Connect}\ e$

The finite nature of the components structure is the intention of the following schema of induction.

[OL-3] *(connectivity induction)*

If ϕ is a property for components, such that:

1. $\forall_x [x\ \mathsf{HasInit}\ c \Rightarrow \phi(c)]$
2. $\forall_d [d\ \mathsf{Connect}\ c \Rightarrow \phi(d)] \Rightarrow \phi(c)$

then $\forall_c [\phi(c)]$.

The construction of Connect from Succ and Decomp is layed down in the following axiom:

[OL-4] *(connection)* $c\,\mathsf{Connect}_1\,d \iff \mathsf{Succ}(c) = d \vee \mathsf{Decomp}(d) = c$

where the direct connection $\mathsf{Connect}_1$ is obtained by the following definition:

$$c\,\mathsf{Connect}_1\,d \equiv c\,\mathsf{Connect}\,d \wedge \neg\exists_e\,[c\,\mathsf{Connect}\,e \wedge e\,\mathsf{Connect}\,d]$$

A first observation is that two components are either in a decomposition relation or a successor relation with each other.

Lemma 5.6

1. $\mathsf{Decomp}(d) = c \Rightarrow \neg\,\mathsf{Succ}(c) = d$
2. $\mathsf{Succ}(d) = c \Rightarrow \neg\,\mathsf{Decomp}(d) = c$

Proof:

1. Suppose $\mathsf{Decomp}(d) = c$. Then we have to prove that $\neg\,\mathsf{Succ}(c) = d$. Assume that $\mathsf{Succ}(c) = d$. From $\mathsf{Decomp}(d) = c$ we can derive that (via the definitions of Connect and $\mathsf{Connect}_1$) that $c\,\mathsf{Connect}\,d$. In the same way from $\mathsf{Succ}(c) = d$, $d\,\mathsf{Connect}\,c$ is derived. Using axioms OL-2 and OL-1 a cyclic structure is found, and so we can conclude $\neg\,\mathsf{Succ}(c) = d$.
2. Analogous to the proof of part 1.

Each component not equal to some initialization component has a so-called *preconnector*.

[OL-5] *(preconnector)* $\neg x\ \mathsf{HasInit}\ c \Rightarrow \exists_d\,[d\,\mathsf{Connect}_1\,c]$

From the connectivity induction axiom the following lemma, describing a one step induction scheme, can be deduced:

Lemma 5.7 Suppose R is a relation on components such that:

1. $\forall_x\,[x\ \mathsf{HasInit}\ c \Rightarrow R(c)]$
2. $\forall_d\,[d\,\mathsf{Connect}_1\,c \Rightarrow R(d)] \Rightarrow R(c)$

Then $\forall_c\,[R(c)]$

Proof:

Suppose R is relation fulfilling the conditions 1 and 2, then we have to prove that $\forall_c\,[R(c)]$. Apply axiom OL-3.

1. It is easily seen that from 1 it can be derived that $\forall_x [x \text{ HasInit } c \Rightarrow R(c)]$
2. Suppose $\forall_d [c \text{ Connect } d \Rightarrow \phi(c)]$ (*). We have to prove that $R(c)$ holds. Suppose, using condition 2, that $d \text{ Connect}_1 c$. Then we know from the definition of Connect_1 that $d \text{ Connect } c$ and thus, using (*), $R(d)$. Applying the result to condition 2 we can conclude $R(c)$.

Thus $\forall_c [R(c)]$.

It will be convenient to introduce the reflexive extension of the connection relation:

$$c \text{ Connect}^* d \equiv c = d \vee c \text{ Connect } d$$

Each component must be effective in the sense that it may be executed in some trace. Thus a component is either a successor of an initialization task of exactly one object type, or it follows up on an initial point of a decomposition alternative of a structural component:

[OL-6] *(executability)* $\exists!_x [x \text{ Owns } c]$

where $x \text{ Owns } c$ is an abbreviation for $d \text{ Connect}^* c$ with $x \text{ HasInit } d$ for some d.

From the set of axioms it should be derivable that the initialization task has no preconnecting components.

Lemma 5.8 $x \text{ HasInit } d \Rightarrow \neg\exists_c [c \text{ Connect } d]$

Proof:

Suppose $x \text{ HasInit } d$ and $c \text{ Connect } d$. Applying OL-6 results in $y \text{ Owns } c$ for some object type y and $x \text{ Owns } d$. As a result of $c \text{ Connect } d$ and $y \text{ Owns } c$ we know that $y \text{ Owns } d$. Applying OL-6 results in $x = y$. Furthermore, since $x \text{ HasInit } d$ and $x \text{ Owns } c$ we know that $d \text{ Connect}^* c$. Applying the definition of Connect^* results in $c = d$ or $d \text{ Connect } c$. Both results conflict with the fact that Connect forms a partial order of components. Thus $x \text{ HasInit } d \Rightarrow \neg\exists_c [c \text{ Connect } d]$.

Obviously, each path of connections of length > 1 must consist of a first step and a last step. The following lemma formalizes this property for the last step.

Lemma 5.9 $c \text{ Connect}^* d \wedge c \neq d \Rightarrow \exists_e [c \text{ Connect}^* e \wedge e \text{ Connect}_1 d]$

Proof:

The proof is based on induction. For abbreviation we will write:
$\phi(d) = \forall_c [c \text{ Connect}^* d \wedge c \neq d \Rightarrow \exists_e [c \text{ Connect}^* e \wedge e \text{ Connect}_1 d]]$

1. Suppose x HasInit d. We have to prove $\phi(d)$. Suppose $c \text{ Connect}^* d \wedge c \neq d$. Using the definition of Connect^* shows $c \text{ Connect } d$. This is in contradiction with lemma 5.8. Thus premise false and as a consequence the implication is true.

2. Suppose $\phi(c) \wedge c\,\mathsf{Connect}\,d$. Furthermore, we assume x HasInit f. We have to prove $\phi(d)$. Suppose $c\,\mathsf{Connect}^*\,d \wedge c \neq d$. Using the definition of Connect* shows $c\,\mathsf{Connect}\,d$. Lemma 5.8 expresses that $d \neq f$ for all x. From axiom OL-5 we know $\exists_e [e\,\mathsf{Connect}\,d]$, or $\exists_e [e\,\mathsf{Connect}^*\,e \wedge e\,\mathsf{Connect}\,d]$. Take $e = c$ will fulfills.

Thus $\forall_c [c\,\mathsf{Connect}^*\,d \wedge c \neq d \Rightarrow \exists_e [c\,\mathsf{Connect}^*\,e \wedge e\,\mathsf{Connect}_1\,d]]$

A repetition component may have at most one decomposition alternative:

[OL-7] *(repetition decomposition)* $c \in \mathcal{X} \wedge \mathsf{Decomp}(d) = c \wedge \mathsf{Decomp}(e) = c \Rightarrow d = e$

Obviously, the course of life of an object type contains at least the predicators of that object type:

[OL-8] *(minimal tasks)* $\mathsf{Actor}(t)\,\mathsf{Owns}\,t$

Finally, we consider objectified action types. The predicators of an objectified action type are involved in the birth of instances of the objectified action type:

[OL-9] *(auto initialization)* $\mathsf{Objectified}(a) \Rightarrow \forall_{p \in a} \exists_c [a\,\mathsf{HasInit}\,c \wedge \mathsf{Pred}(c) = p]$

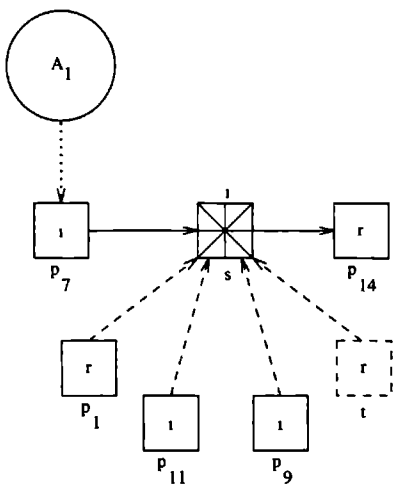


Figure 5.7: Object life model

Example 5.11

Consider the object life model displayed in figure 5.7 which is an abstract view on the object life model of *Band* displayed in figure 4.7. This model is described by:

$\mathcal{T}$	$\mathcal{D}$	$\mathcal{X}$	$\mathcal{Y}$	$\mathcal{Z}$
p_1	t	$\imath$	s	
p_7				
p_9				
p_{11}				
p_{14}				
t				

Succ	$\mathcal{C}$
p_7	$\imath$
$\imath$	p_{14}

Decomp	$\mathcal{C} \setminus \mathcal{T}$
s	i
p_1	s
p_9	s
p_{11}	s

HasInit
A_1 HasInit p_7

Pred	$\mathcal{P}$
t	p_4

5.4.2 Traces of object life model

An object life model can be transformed into a *process algebra* ([BW90a]) expression describing a formal semantics for the course of life of its corresponding object type (see section 5.4.2.1). Expressions in process algebra can be rewritten, using a number of rewrite rules (see appendix E), showing semantic equivalence of object life models (see section 5.4.2.2). Furthermore, an expression in process algebra can be seen as a generator of *traces*. Section 5.4.2.3 discusses an example of how the so-called *object trace space* is obtained from a process algebra expression.

Each process algebra expression describes a narrowed view on the course of life of its corresponding object type, i.e. it does not necessarilly describe parts of the courses of life of type related object types. The object trace space of an object type is derived from its own object life model and that of its type related object types. The object trace space of an object type describes *histories* for its objects. However, the object trace space can be defined to wide, i.e. it allows invalid histories. In section 5.4.2.4 the relation between object trace spaces and object histories are described leading to a number of domain independent restriction rules on object histories. Domain dependent restriction rules on object histories are discussed in section 5.4.2.5. Such restriction rules are also referred to as *property functions* or *constraints*. Finally, in section 5.4.2.6, a special property function is discussed which can compute the state of instances for a particular history.

5.4.2.1 Trace generator

The narrowed course of life of object type x can be defined as

$$E_x = \bigoplus_{x \text{ HasInit } c} E_c$$

The semantics of component c is denoted as E_c. This function will be recursively defined in the sequel of this section. For each component c a translation to process algebra equations is defined:

$$E_c = \begin{cases} \mathsf{Exec}(c) & \text{if } c \text{ has no successor} \\ \mathsf{Exec}(c) \odot E_{\mathsf{Succ}(c)} & \text{otherwise} \end{cases}$$

The execution of a single component is described by the function Exec. We distinguish the following cases:

1. The execution of task t consists of the execution of its corresponding action type in the role specified by predicator t. Thus:

$$\mathsf{Exec}(t) = t$$

2. The execution of a surrogate q coincides with that of its associated predicator:

$$\mathsf{Exec}(q) = \mathsf{Exec}(\mathsf{Pred}(q))$$

3. The execution of a choice s consists of the execution of one of its alternatives:

$$\mathsf{Exec}(s) = \bigoplus_{\mathsf{Decomp}(c)=s} E_c$$

4. A repetition symbol i leads to a repeated execution of its body. Let c be the unique decomposition of i, i.e. $\mathsf{Decomp}(c) = i$. Then:

$$\mathsf{Exec}(i) = \bigoplus_{n \geq 0} E_c^n = E_c^*$$

Note that $E_c^0 = \varepsilon$.

5. The last symbol to be defined is the merge symbol p:

$$\mathsf{Exec}(p) = \bigparallel_{\mathsf{Decomp}(c)=p} E_c$$

Example 5.12

The semantics in terms of example 5.11 is defined as follows:

$$\begin{array}{lcl} E_{A_1} & = & E_{p_7} \\ E_{p_7} & = & p_7 \odot E_i \\ E_i & = & E_s^* \odot E_{p_{14}} \\ E_s & = & E_{p_1} \oplus E_{p_9} \oplus E_{p_{11}} \oplus E_t \\ E_{p_1} & = & p_1 \\ E_{p_9} & = & p_9 \\ E_{p_{11}} & = & p_{11} \\ E_t & = & p_4 \\ E_{p_{14}} & = & p_{14} \end{array}$$

The resulting process algebra equation for the course of life of object type *Band* is:

$$E_{A_1} = p_7 \odot (p_1 \oplus p_9 \oplus p_{11} \oplus p_4)^* \odot p_{14}$$

5.4.2.2 Life equivalence

An advantage of process algebra is that it contains rewrite rules which can be used to prove semantic equivalence. For example consider the courses of life of figure 5.8. The course of life of object type X is described by both $E_X = a \odot (b \oplus (c \oplus b))$ and $E_X = a \odot (b \oplus c)$. By applying rewriting rules of appendix E the equivalence of these courses of life can be shown (see example 5.13).

Example 5.13

$$\begin{aligned} a \odot (b \oplus (c \oplus b)) &= \{\text{BPA1}\} \\ a \odot ((c \oplus b) \oplus b) &= \{\text{BPA2}\} \\ a \odot (c \oplus (b \oplus b)) &= \{\text{BPA3}\} \\ a \odot (c \oplus b) &= \{\text{BPA1}\} \\ a \odot (b \oplus c) & \end{aligned}$$

However, there are some courses of life that are observational the same which cannot be proven equivalent within the system of axioms. For example, the process algebra equation $X \odot (Y \oplus Z) = X \odot Y \oplus X \odot Z$ does not hold for process variables in general and is as a consequence not an axiom of the family of algebras. However, for concrete atomic processes the equation is correct. The theory of *bisimulation* ([BW90a]) offers an extended notion of equivalence which can deal with such courses of life.

5.4.2.3 Object trace space

As shown above the semantics of the life of an object type x is expressed by an algebraic expression E_x. The set $\mathsf{Traces}(E_x)$, or $\mathsf{Traces}(x)$ for short, denotes all possible traces of tasks belonging to object type x and is called the *object trace space* of x. The set of all possible traces of all tasks is denoted as $(\mathcal{T})^* \cup \{\lambda\}$ with λ denoting the empty trace. Axioms with respect to the object trace space are defined in table E.5 of appendix E.

Example 5.14

Consider the left part of figure 5.8. The object trace space $\mathsf{Traces}(x)$ is calculated as follows:

$$\begin{aligned} \mathsf{Traces}(x) &= \{\text{definition of } x\} \\ \mathsf{Traces}(a \odot (b \oplus (c \oplus b))) &= \{\text{TR3}\} \\ \{\lambda\} \cup \{a \cdot \sigma \mid \sigma \in \mathsf{Traces}(b \oplus (c \oplus b))\} &= \{\text{TR4}\} \end{aligned}$$

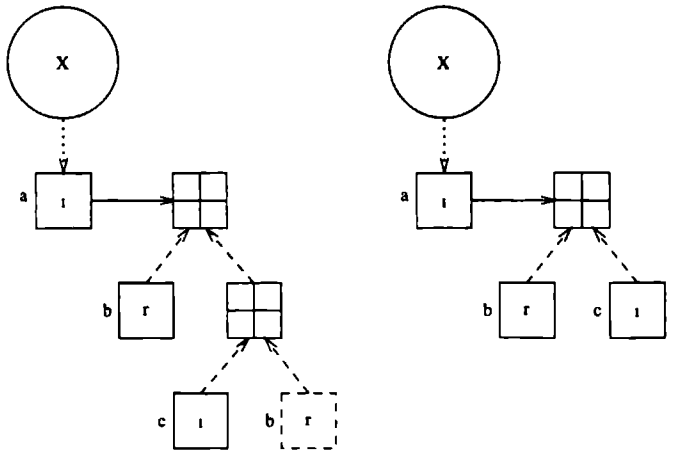


Figure 5.8: Two equivalent courses of life

$$
\begin{aligned}
\{\lambda\} \cup \{a \cdot \sigma \mid \sigma \in \mathsf{Traces}(b) \cup \mathsf{Traces}(c \oplus b)\} &= \{\text{TR4}\} \\
\{\lambda\} \cup \{a \cdot \sigma \mid \sigma \in \mathsf{Traces}(b) \cup \mathsf{Traces}(c) \cup \mathsf{Traces}(b)\} &= \left\{\begin{array}{l}\text{commutativity and} \\ \text{idempotence } \cup\end{array}\right\} \\
\{\lambda\} \cup \{a \cdot \sigma \mid \sigma \in \mathsf{Traces}(b) \cup \mathsf{Traces}(c)\} &= \{\text{TR2 (twice)}\} \\
\{\lambda\} \cup \{a \cdot \sigma \mid \sigma \in \{\lambda, b\} \cup \{\lambda, c\}\} &= \{\text{definition } \cup\} \\
\{\lambda\} \cup \{a \cdot \sigma \mid \sigma \in \{\lambda, b, c\}\} &= \{\text{substitution } \sigma\} \\
\{\lambda\} \cup \{a, a \cdot b, a \cdot c\} &= \{\text{definition } \cup\} \\
\{\lambda, a, a \cdot b, a \cdot c\}
\end{aligned}
$$

As both object life models of figure 5.8 are shown in example 5.13 to be equivalent, the corresponding object traces spaces should be the same.

$$
\begin{aligned}
\mathsf{Traces}(x) &= \{\text{definition of } x\} \\
\mathsf{Traces}(a \odot (b \oplus c)) &= \{\text{TR3}\} \\
\{\lambda\} \cup \{a \cdot \sigma \mid \sigma \in \mathsf{Traces}(b \oplus c)\} &= \{\text{TR4}\} \\
\{\lambda\} \cup \{a \cdot \sigma \mid \sigma \in \mathsf{Traces}(b) \cup \mathsf{Traces}(c)\} &= \{\text{TR2 (twice)}\} \\
\{\lambda\} \cup \{a \cdot \sigma \mid \sigma \in \{\lambda, b\} \cup \{\lambda, c\}\} &= \{\text{definition } \cup\} \\
\{\lambda\} \cup \{a \cdot \sigma \mid \sigma \in \{\lambda, b, c\}\} &= \{\text{substitution } \sigma\} \\
\{\lambda\} \cup \{a, a \cdot b, a \cdot c\} &= \{\text{definition } \cup\} \\
\{\lambda, a, a \cdot b, a \cdot c\}
\end{aligned}
$$

Until now *recursive equations* and *recursive specifications* were not considered. An example of a recursive equation is $X = X \odot a$ and an example of a recursive specification is $E = \{X = X \odot a, Y = a \odot X\}$. The trace axioms of table E.5 are not sufficient to deal with recursive specifications. However, in [BW90a] a number of definitions and axioms are presented which deal with a large number of recursive specifications.

5.4.2.4 Object histories

Let L be a logbook of some object action involvement model and object property model. Then this logbook can be transformed into a population of the meta-model of logbooks (see figure 4.1). This population provides a detailed description of the *history* of the UoD. For our purposes, an abstract view on this history is sufficient. This abstract view is obtained by cumulating all actions that have been occurred:

$$\mathsf{History}(\mathsf{L}) = \bigcup_{a \in \mathcal{A}} \mathsf{L}(a)$$

Thus, generally, a history is a subset from $\mathsf{Time} \times \Omega$. The history of an instance v is obtained by restricting to those actions in which v plays a role:

$$\mathsf{History}(v, \mathsf{L}) = \left\{ \langle t, f \rangle \in \mathsf{History}(\mathsf{L}) \;\middle|\; \exists_p [\pi_{\mathsf{i}}(f(p)) = v] \right\}$$

At each moment, an instance can be involved in at most one action:

[L-1] *(indivisible atomic actions)* $a, b \in \mathsf{History}(v, \mathsf{L}) \wedge \pi_{\mathsf{t}}(a) = \pi_{\mathsf{t}}(b) \Rightarrow a = b$

As a result, the history $\mathsf{History}(v, \mathsf{L})$ can be considered as a sequence of actions. Let $\mathsf{Head}(\mathsf{History}(v, \mathsf{L}))$ be the action first occurring in history $\mathsf{History}(v, \mathsf{L})$. Then the expression $\mathsf{Head}(\mathsf{History}(v, \mathsf{L}))$ must be the birth action of v. With each action in $\mathsf{History}(v, \mathsf{L})$ the function:

$$\mathsf{Preds} : \Omega \times \mathsf{History}(v, \mathsf{L}) \rightarrow \wp(\mathcal{P})$$

associates the predicator(s) corresponding to the role(s) being played by v in this action.

$$\mathsf{Preds}(v, \langle t, f \rangle) = \left\{ p \in \mathcal{P} \;\middle|\; \pi_{\mathsf{i}}(f(p)) = v \right\}$$

Note that $\mathsf{Preds}(v, \langle t, f \rangle)$ may contain more than one predicator. However, an object can not be multiply involved in its birth action.

[L-2] *(passive birth)* $|\mathsf{Preds}(\mathsf{Head}(\mathsf{History}(v, \mathsf{L})))| = 1$

The root type of v is defined as the actor of this single predicator. This object type is denoted as $\mathsf{Type}(v)$. During its life, an instance may become (temporarily) a member of specializations and generalizations.

Next we derive all sequences of predicators describing the life of v. This set of sequences is denoted as $\mathsf{Traces}(v, \mathsf{L})$ and equals the concatenation of the actions in the history of v ($\mathsf{Concat}(\mathsf{History}(v, l))$). The function Concat is defined by:

$$\mathsf{Concat}(\mathsf{L}) = \begin{cases} \lambda & \text{if L is empty} \\ \mathsf{Perms}(\mathsf{Preds}(v, a)) \cdot \mathsf{Concat}(\mathsf{Tail}(\mathsf{L})) & \text{if } a = \mathsf{Head}(\mathsf{L}) \end{cases}$$

where $\mathsf{Perms}(R)$ yields the set of all traces of predicators in R. The life of instance v should be in accordance with the object trace space from its root type, its specializations and its generalizations. Suppose instance v has root type $x = \mathsf{Type}(v)$.

[L-3] *(root life)* $\mathsf{Restrict}(\mathsf{Traces}(v, \mathsf{L}), x) \in \mathsf{Traces}(x)$

The function $\mathsf{Restrict}(\omega, x)$ restricts trace ω to object type x and is inductively defined by:

$$\begin{aligned} \mathsf{Restrict}(\lambda, x) &= \lambda \\ \mathsf{Restrict}(p \cdot \omega, x) &= \begin{cases} p \cdot \mathsf{Restrict}(\omega, x) & \text{if } \mathsf{Actor}(p) = x \vee \mathsf{Actor}(\mathsf{Pred}(p)) = x \\ \mathsf{Restrict}(\omega, x) & \text{otherwise} \end{cases} \end{aligned}$$

Instances of object type x can be a member of each specialization of x. The next rule describes the life from the perspective of a specialization.

[L-4] *(specialized life)* $y \,\mathsf{spec}\, x \Rightarrow \mathsf{Restrict}(\mathsf{Traces}(v, \mathsf{L}), y) \in \mathsf{Traces}^*(y)$

The following rule is the analogon for generalization.

[L-5] *(generalized life)* $y \,\mathsf{gen}\, x \Rightarrow \mathsf{Restrict}(\mathsf{Traces}(v, \mathsf{L}), y) \in \mathsf{Traces}^*(y)$

Furthermore, the rules for being member of subtypes and generalizations must be obeyed. Each specialization and each generalization has associated a predicate stating its extentionality. Let $\mathsf{SubRule}_x$ be the membership predicate for a specialized object type x. Then instance v is a member of x iff the expression $\mathsf{SubRule}_x(v, \mathsf{L})$ is true.

[L-6] *(valid membership (1))*
$\langle t, f \rangle \in \mathsf{History}(v, \mathsf{L}) \wedge p \in \mathsf{Preds}(v, \langle t, f \rangle) \wedge \mathsf{IsSpec}(\mathsf{Actor}(p)) \Rightarrow \mathsf{SubRule}_{\mathsf{Actor}(p)}(v, \mathsf{History}(t, \mathsf{L}))$

where $\mathsf{History}(t, \mathsf{L})$ provides a snapshot of L at time t. An analogous requirement should hold for generalized object types.

[L-7] *(valid membership (2))*
$\langle t, f \rangle \in \mathsf{History}(v, \mathsf{L}) \wedge p \in \mathsf{Preds}(v, \langle t, f \rangle) \wedge \mathsf{IsGen}(\mathsf{Actor}(p)) \Rightarrow \mathsf{GenRule}_{\mathsf{Actor}(p)}(v, \mathsf{History}(t, \mathsf{L}))$

5.4.2.5 Restricting object histories

The object life model provides a framework for histories of instances of each object type. However, this demarcation may be too wide. Further restrictions, also called *history constraints*, can be described in terms of the properties of objects. Such restrictions may involve objects of several types. Restrictions on histories can be defined with so-called *property functions*.

Property functions assign values to histories. These values can be taken from Ω. The value $\perp$ (undefined) is used to make property functions total functions.

Example 5.15

Reconsider the course of life of a *Band* (see figure 4.7). Suppose $b \in \mathsf{L}(\text{Band})$ and $p, q \in \mathsf{L}(\text{Person})$. The following histories for bands obviously are not valid (for readability, time stamps are omitted, and action type names are added; furthermore we denote histories in a sequence notation as in programming languages):

h_1: *to set up*(b,p); *to leave*(b,q)

h_2: *to set up*(b,p); *to join*(b,p); *to join*(b,p)

The problem with these histories is that a person can only leave a band after joining this band. This constraint is referred to as BM1. Furthermore, only new members can join a band. This constraint is referred to as BM2. In order to express these constraints, we introduce the property function

$$\mathsf{BandMembers} : \left\{ \mathsf{History}(b, \mathsf{L}) \;\middle|\; \mathsf{L} \text{ a logbook} \right\} \rightarrow \Omega$$

as follows:

$$\begin{aligned}
\mathsf{BandMembers}(\langle\rangle) &= \varnothing \\
\mathsf{BandMembers}(\textit{to set up}(b,p)) &= \{p\} \\
\mathsf{BandMembers}(h; \textit{to join}(b,p)) &= \begin{cases} \perp & \text{if } p \in \mathsf{BandMembers}(h) \\ \mathsf{BandMembers}(h) \cup \{p\} & \text{otherwise} \end{cases} \\
\mathsf{BandMembers}(h; \textit{to leave}(b,p)) &= \begin{cases} \perp & \text{if } p \notin \mathsf{BandMembers}(h) \\ \mathsf{BandMembers}(h) - \{p\} & \text{otherwise} \end{cases} \\
\mathsf{BandMembers}(h; \textit{to disband}(b)) &= \varnothing
\end{aligned}$$

where $\langle\rangle$ denotes the empty history, and h stands for any history of *Band* b. This leads to the following formulation for both BM1 and BM2:

$$\mathsf{BandMembers}(h) \neq \perp$$

The verbalization of such constraints requires a language to formulate constraints. For this purpose we use the language Elisa-D ([PW95b]).

```
LET is_member_of BE
  ALL Person joins Band EVER MINUS ALL Person leaves Band EVER
```

The construction LET ... BE ... is a mechanism to extend the information grammar with new rules for forming sentences. The construction ALL P EVER gathers all instances of information descriptor P from the past. The constraints BM1 and BM2 can be expressed as:

```
CONSTRAINT
  BM1: NOT Person leaves Band BUT NOT is_member-of THAT Band
  BM2: NOT Person joins Band AND ALSO is_member-of THAT Band
```

5.4.2.6 Evaluating state records

In section 5.3.4 the concept of state record has been introduced as the set of all properties of an object type. In this section we show how values of properties of instances can be computed from histories.

The state record of an instance v after logbook L contains those properties which are introduced by the types of v. The state of an instance v at the end of a logbook L can be defined by the property function:

$$\mathsf{State}(v, \mathsf{L}) : \left\{ p \;\middle|\; \exists_x \left[\mathsf{IsDefiner}(\mathsf{L}, v, \mathsf{ANm}(\mathsf{Update}(p)), x) \right] \right\} \rightarrow \Omega$$

with $\mathsf{State}(v, \mathsf{L})(p) = \mathsf{Eval}(\mathsf{History}(v, \mathsf{L}), p)$, where the function Eval is defined in the following way:

$$\begin{array}{rcll} \mathsf{Eval}(\langle\rangle, p) & = & \bot & \\ \mathsf{Eval}(h; \langle t, f\rangle, p) & = & \mathsf{Eval}(h, p) & \text{if } \langle t, f\rangle \notin \mathsf{L}(\mathsf{Update}(p)) \\ \mathsf{Eval}(h; \langle t, \langle v, m\rangle\rangle, p) & = & m & \text{if } \langle t, \langle v, m\rangle\rangle \in \mathsf{L}(\mathsf{Update}(p)) \end{array}$$

5.4.3 Extended paraphrasing mechanism

Paraphrasing the object life model is not straightforward as it requires not a single sentence. In the rest of this section we give some rules of the thumb. Paraphrasing object life models is still a topic for research (see also [Dal95]).

5.4.3.1 Meta rules

The final driver for executing the information grammar should be able to focus on paraphrasing the lifes of its object types. Therefore, the domain independent affix rule ⟨mode⟩ is extended with an affix life, i.e.

⟨mode⟩ :: actual | structural | trigger | life

As a result the procedure 'Independent Meta Rules' is also modified.

5.4.3.2 Course of life

The life of an object type starts with its initialization, followed by a sequence of components. Suppose object type x starts its life with component c. This leads to the following grammar rule:

$$\langle \text{life of } x \rangle : \langle \text{seq } c \rangle$$

A sequence of components is elaborated as follows:

$$\langle \text{seq } c \rangle : \begin{cases} \langle c \rangle \text{ "." "Then" } \langle \text{seq } d \rangle & c \,\mathsf{Connect}_1\, d \\ \langle c \rangle \text{ "."} & \text{otherwise} \end{cases}$$

Before paraphrasing rules for components are introduced an auxiliary predicate InDisplay is defined. $\mathsf{InDisplay}(x, [D])$ states whether affix (non)terminal x occurs in the display D of nonterminal $\langle \mathsf{FNm}(a)\,[D] \rangle$. For example, predicate $\mathsf{InDisplay}(r, [\text{r,Recording,Producer}])$ yields true.

We distinguish the following cases for component c:

1. Suppose c is a predicator, i.e. $c \in \mathcal{P}$. If there are no other object types using a surrogate of c, $\neg\exists_{y,d}\,[x \neq y \wedge \mathsf{Pred}(d) = c \wedge y\ \mathsf{Owns}\ d]$, the following rule is added:

   $$\langle c \rangle : \langle \mathsf{FNm}(\mathsf{Action}(c)) \rangle$$

   In case there are surrogates of c used in the course of life of other object types, a more precise rule is necessary:

   $$\langle c \rangle : \langle \mathsf{FNm}(\mathsf{Action}(c))\,[D] \rangle$$

   such that $\mathsf{InDisplay}(\mathsf{FNm}(x), [D])$.

2. Suppose c is a surrogate, i.e. $c \in \mathcal{D}$. If there are no other object types using a surrogate of its corresponding predicator, $\neg\exists_{y,d}\,[x \neq y \wedge \mathsf{Pred}(d) = \mathsf{Pred}(c) \wedge y\ \mathsf{Owns}\ d]$, the following rule is added:

   $$\langle c \rangle : \langle \mathsf{FNm}(\mathsf{Action}(\mathsf{Pred}(c))) \rangle$$

   Otherwise the following (more sophisticated) rule is required:

   $$\langle c \rangle : \langle \mathsf{FNm}(\mathsf{Action}(\mathsf{Pred}(c)))\,[D] \rangle$$

   such that $\mathsf{InDisplay}(\mathsf{FNm}(x), [D])$.

3. Suppose c is a repetition predicator, i.e. $c \in \mathcal{X}$. Furthermore, component d is a decomposition of c, i.e. $\mathsf{Decomp}(d) = c$. This leads to the following rule:

   $$\langle c \rangle : \text{"repeatedly" } \langle \text{frag } d \rangle$$

4. Suppose c is a choice predicator, i.e. $c \in \mathcal{Y}$. Furthermore, components $d_1, \ldots, d_k$ are decompositions of c. This leads to the following rule:

   $\langle c \rangle$: $\langle$frag $d_1\rangle$ "or" ... "or" $\langle$frag $d_k\rangle$

5. Suppose c is a merge predicator, i.e. $c \in \mathcal{Z}$. Furthermore, components $d_1, \ldots, d_k$ are decompositions of c. This leads to the following rule:

   $\langle c \rangle$: "interleaved" $\langle$frag $d_1\rangle$ "or" ... "or" $\langle$frag $d_k\rangle$

The nonterminal $\langle$frag $d\rangle$ leads to the description of the course of life starting from components d. If the component d is not followed by another component, i.e. $\neg\exists_d\,[e\ \mathsf{Connect}_1\ d]$, no special care is required. Otherwise, the description leads to a text fragment within some other sentence. Therefore, this is marked explicitly with brackets:

$$\langle \text{frag } d\rangle : \begin{cases} \text{"["}\ \langle \text{seq } d\rangle\ \text{"]"} & \text{if } d \text{ has a successor} \\ \langle d\rangle & \text{otherwise} \end{cases}$$

For each object type the paraphrasing rules for its course of life are collected using the following procedures in pseudo code:

```
proc Paraphrase Object Life (O x) : G_inf rules ;
   return {⟨life of FNm(x)⟩ : ⟨seq c⟩ | x HasInit c} ∪ Paraphrase Course of Life(c)
endproc

proc Paraphrase Course of Life (C c) : G_inf rules ;
   if ∃_d [c Connect_1 d]
   then return {⟨seq c⟩ : ⟨c⟩ "." "Then" ⟨seq d⟩ } ∪ Paraphrase Course of Life(d)
   else return {⟨seq c⟩ : ⟨c⟩ "."} ∪ Paraphrase Component(c)
   endif
endproc

proc Paraphrase Component (C c) : G_inf rules ;
   return
      case
         c ∈ P :
            if ¬∃_{y,d} [Actor(c) ≠ y ∧ Pred(c) = d ∧ y Owns d]
            then {⟨c⟩ : ⟨FNm(Action(c))⟩ }
            else {⟨c⟩ : ⟨FNm(Action(c)) [D]⟩ | InDisplay(FNm(Actor(c)), [D])}
            endif
         c ∈ D :
            if ¬∃_{y,d} [Actor(Pred(c)) ≠ y ∧ Pred(d) = Pred(c) ∧ y Owns d]
            then {⟨c⟩ : ⟨FNm(Action(Pred(c)))⟩ }
            else {⟨c⟩ : ⟨FNm(Action(Pred(c))) [D]⟩ | InDisplay(FNm(Actor(Pred(c))), [D])}
            endif
         c ∈ X :
            {⟨c⟩ : "repeatedly" ⟨frag d⟩ | Decomp(d) = c} ∪
            Paraphrase Fragment(d)
```

$c \in \mathcal{Y}$:
$\{\langle c\rangle : \langle \text{frag } d_1\rangle$ "or" ... "or" $\langle \text{frag } d_k\rangle \mid \text{Decomp}(d_i) = c\} \cup \bigcup_{i=1}^{k} \text{Paraphrase Fragment}(d_i)$
$c \in \mathcal{Z}$:
$\{\langle c\rangle :$ "interleaved" $\langle \text{frag } d_1\rangle$ "or" ... "or" $\langle \text{frag } d_k\rangle \mid \text{Decomp}(d_i) = c\} \cup \bigcup_{i=1}^{k} \text{Paraphrase Fragment}(d_i)$
endcase
endproc

proc Paraphrase Fragment ($\mathcal{C}$ c) : G_{inf} rules ;
if $\exists_d$ [Succ(c) = d]
then return $\{\langle \text{frag } c\rangle :$ "[" $\langle \text{seq } c\rangle$ "]" $\} \cup$ Paraphrase Course of Life(c)
else return $\{\langle \text{frag } c\rangle : \langle c\rangle\} \cup$ Paraphrase Component(c)
endif
endproc

As discussed in section 4.3, only the courses of life of non-label types and non-action types, i.e. the active object types, are interesting for paraphrasing. For each module the courses of life of all active object types are collected using the following procedure in pseudo code:

proc Paraphrase Object Lifes ($\mathcal{M}$ m) : G_{inf} rules ;
return $\left\{\langle \text{FNm}(m)\,[\text{life}]\rangle : +\!\!+_{x \in \text{Locals}(m) \cap \mathcal{I}} \langle \text{life of FNm}(x)\rangle\right\} \cup \bigcup_{x \in \text{Locals}(m) \cap \mathcal{I}} \text{Paraphrase Object Life}(x)$
endproc

5.4.3.3 Extended driver

The information grammar is extended by paraphrasing rules for the courses of life of object types. As a result the procedure 'Paraphrase Module' has to be modified slightly by adding 'Paraphrase Object Lifes(m)'. Furthermore, the start rule for paraphrasing a module:

$$\left\{\langle \text{FNm}(m)\,[\text{mode}]\rangle : \langle \text{FNm}(m)\,[\text{actual}]\rangle \mid \langle \text{FNm}(m)\,[\text{structural}]\rangle \mid \langle \text{FNm}(m)\,[\text{trigger}]\rangle\right\}$$

is transformed into:

$$\left\{\langle \text{FNm}(m)\,[\text{mode}]\rangle : \langle \text{FNm}(m)\,[\text{actual}]\rangle \mid \langle \text{FNm}(m)\,[\text{structural}]\rangle \mid \langle \text{FNm}(m)\,[\text{trigger}]\rangle \mid \langle \text{FNm}(m)\,[\text{life}]\rangle\right\}$$

Executing nonterminal $\langle \text{Main}\,[\text{life}]\rangle$ leads to a textual description of all courses of life of the active object types in the UoD.

5.5 Summary and outlook

In this chapter a conceptual object-oriented model has been described, called the information architecture, which is a composition of three object-oriented analysis models. The

information architecture provides a structural description of the logbook (normal form specification). For each analysis model there has been described (1) a formal syntax and semantics, and (2) algorithms to obtain the information grammar. Furthermore, the integration between these three analysis models has been discussed. Finally, we have shown that the information architecture describes possible histories for objects, i.e. the information architecture can describe a concrete logbook.

In chapter 6 verification and validation aspects with respect to the information architecture and information grammar are presented. An in-depth treatment of the semantics of the object action involvement model and the object property model is provided in chapter 7 using a framework which is based on category theory.

Chapter 6

Validation, Verification, and Design

I said I know it's only rock 'n' roll but I like it
I said I know it's only rock 'n' roll but I like it
But I like it I like it yes I do

From: "It's Only Rock 'n' Roll",
The Rolling Stones

6.1 Introduction

In this chapter[1] the way of supporting and validating is discussed. Furthermore, some verification and design issues are presented with respect to information architectures.

In validating the information grammar (and thus the information architecture) we make use of the AGFL formalism and system. Both formalism and system are introduced in section 6.2. In section 6.3 the validation of the information grammar is demonstrated by generating and parsing sample sentences.

In chapter 5 axioms have been introduced to enforce correct analysis models. However, it is still possible that a syntactically correct model can not be instantiated. In section 6.4 the so-called *life dependency graph*, which is a special view on the object action involvement model, is introduced to detect a number of non-populatable models.

After verification and validation of an information architecture a first milestone of the development process is achieved. A next milestone in the development process is the design of an information architecture. In section 6.5 it is outlined how an object-oriented design of the information architecture can be obtained. Finally, in section 6.6 a summary of this chapter and an outlook to the next chapters is presented.

[1]This chapter is based on [DFW96], [FW96c], and [FW96e].

6.2 AGFL formalism and system

The AGFL (Affix Grammars over Finite Lattices) formalism ([Kos91],[Kos96]) has been developed for the description of natural languages In this section a brief introduction to the AGFL formalism and its implementation is given

6.2.1 Formalism

Affix Grammars are a family of two-level grammars where the first level consists of context-free *production rules* augmented with *affixes* and the second level defines the *domains* of these affixes AGFLs are a particularly simple form of Affix Grammars, in which restricted context-free *affix rules* generate a finite domain for each affix

In order to introduce AGFL we need some terminology An affix rule consists of a *nonterminal affix* followed by a double colon followed by a series of *affix alternatives* separated by semicolons followed by a dot An affix alternative consists of either a *terminal affix* or a nonterminal affix

A production rule consists of a *nonterminal* followed by a single colon followed by a series of *alternatives* again separated with semicolons and followed by a dot An alternative consists of a possibly empty series of *members* separated with commas A member is either a nonterminal or a terminal enclosed in quotes A nonterminal consists of a *head* and an optional *display* A display is a series of *affix expressions* separated with commas and enclosed in braces An affix expression is either a nonterminal affix or a concrete set of terminal affixes

The following grammar shows an affix rule with two alternatives and two production rules with one alternative each

```
R :: 1; r.

TO PRODUCE(r) . PRODUCER, "produces", RECORDING

TO PRODUCE(1) : RECORDING, "is produced by", PRODUCER.
```

Nonterminal `TO PRODUCE(r)` consists of a head `TO PRODUCE` and display `(r)`

Given some *start* nonterminal, sentences are produced by repeatedly replacing a nonterminal with one of its alternatives In AGFL, affixes are used for enforcing agreement between parts of speech The *consistent substitution rule* demands that in an alternative all occurrences of the same affix must obtain the same value This value should be a non-empty set of terminal affixes for which rewriting succeeds Using affixes in order to express various constraints such as uniqueness, cardinality, etcetera and representing object class hierarchies with affix rules is still an interesting subject of investigation

In AGFL, affix rules determine power-set lattices over the affix domains with intersection as only operation The lattice for the affix `R` is shown in figure 6 1 The top element `R` of

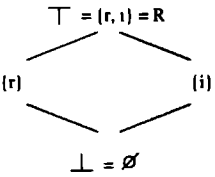


Figure 6.1: The lattice of values for the affix R

the lattice can be seen as the union of all possibilities (the values may be r or i, or their combination). During rewriting the set of possible values of an affix may be narrowed down to a smaller set by intersection with other affixes. However, an affix may not assume the empty set, the bottom element of the lattice, as value.

The generative power of AGFLs does not exceed the power of context-free grammars. However, AGFLs have better descriptive properties and allow quite efficient implementation. In practice, the limitations imposed on AGFL do not turn out to be too restrictive for describing the syntax of natural languages ([KO96]).

6.2.2 System

The AGFL system consists of a collection of software systems for natural language processing and grammar development. Here, we will describe the *grammar workbench* (GWB) and the GEN *parser generator*.

The grammar workbench is an environment for the development of grammars in the AGFL formalism. Its purposes are to allow the user to edit grammars, to perform consistency checks on a grammar, to compute grammar properties, to assist in performing grammar transformations and to generate sample sentences. We are particularly interested in the GWBs capability to generate sample sentences from a grammar in AGFL. The ideas behind the grammar workbench and its implementation are elaborated in [DKNZ92b] and [DKNZ92a].

The GEN parser generator[2] generates a parser for a possibly ambiguous or left recursive grammar in the AGFL formalism. Real life applications require a grammar to use large lexical data bases (*lexica*) in an efficient way. Therefore, GEN optionally invokes its integrated lexicon system which compresses a lexicon and builds an index for fast access. During lexical analysis the parsers generated by GEN try to identify the categories of each word in an input sentence in the lexicon. If lexical analysis succeeds, the parser will generate all parse trees according to the grammar. These parse trees can be translated (transduced) to other representations depending on the application domain. For instance, one could experiment with transduction of queries in restricted natural language to expressions in the query language LISA-D ([HPW93]).

[2]The AGFL system and its description can be obtained at the AGFL www-site at http://www.cs.kun.nl/agfl/.

6.3 Validating information grammars

In this section a demonstration of the sample information grammar of appendix D is presented using the AGFL system[3]. The system is presented using a Windows 95 look and feel user-interface[4].

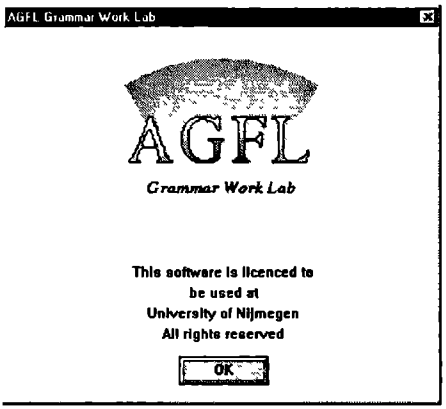


Figure 6.2: Introduction window AGFL system

Figure 6.2 shows the introduction window of the AGFL Grammar Work Lab. The file `exp.gra` of appendix D is read (see figure 6.3). The properties of `exp.gra` are presented in the right side of the window. Furthermore, the (affix) nonterminals of the grammar are presented in the two window boxes at the left. Once a grammar has been read a number of options are available. For example, (affix) nonterminals can be inspected and edited. However, the main actions are:

1. generation of sentences of a particular nonterminal by focusing (clicking) on that nonterminal (button *Generate Corpus*), and
2. generation of a parser for the grammar read (button *Generate Parser*).

6.3.1 Sentence generation

Figure 6.4 shows two different textual productions generated from the nonterminal `LIFE OF BAND` currently in focus:

[3]The AGFL system was developed by the following persons (in alphabetic order): Caspar Derksen, Franc Grootjen, Paul Jones, Arjan Knijff (NWO/NFI), Mark-Jan Nederhof (NWO/NFI), and Arend van Zwol (NWO/SION).

[4]This user-interface was implemented by Rob Bosman.

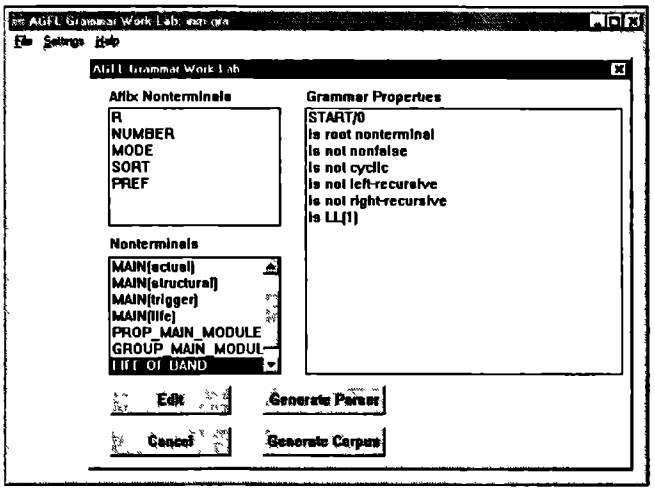


Figure 6.3: Reading grammar `exp.gra`

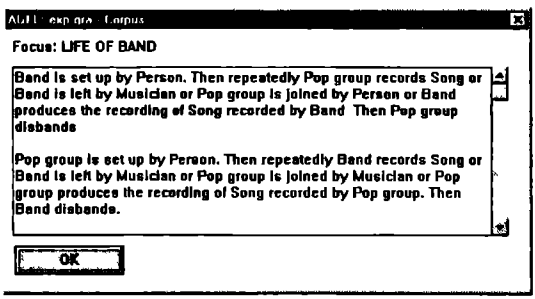


Figure 6.4: Generated corpus for nonterminal `LIFE OF BAND`

```
LIFE OF BAND : SEQ_C_I.

SEQ_C_I : TO SET UP(i), ".",
          THEN, SEQ_D_I.

SEQ_D_I : REPEATEDLY, FRAG_D_I, OR, FRAG_D_II, OR,
                      FRAG_D_III, OR, FRAG_D_IV, ".",
          THEN, SEQ_D_II.

SEQ_D_II : TO DISBAND, ".".

FRAG_D_I   : TO RECORD(r,song,band).
```

```
FRAG_D_II  : TO LEAVE(i).
FRAG_D_III : TO JOIN(i).
FRAG_D_IV  : TO PRODUCE(r,recording,band).
```

Note that in the information grammar of appendix D we have included two ways of paraphrasing object type Band: *Band* and *Pop group*.

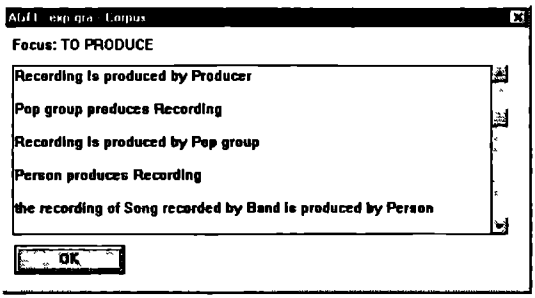


Figure 6.5: Generated corpus for nonterminal TO PRODUCE

It is also possible to focus on a particular action type. Figure 6.5 shows a number of different textual descriptions for action type *to produce*. The nonterminal TO PRODUCE with rules:

```
TO PRODUCE : TO PRODUCE(r) ; TO PRODUCE(i).
TO PRODUCE(r) : TO PRODUCE(r,recording,producer) ;
                TO PRODUCE(r,recording,band) ;
                TO PRODUCE(r,recording,person) ;
                TO PRODUCE(r,recording,musician).
TO PRODUCE(i) : TO PRODUCE(i,recording,producer) ;
                TO PRODUCE(i,recording,band) ;
                TO PRODUCE(i,recording,person) ;
                TO PRODUCE(i,recording,musician).
```

allows the generation of sentences for all object types which bind the corresponding action type, being *Band*, *Musician*, *Person*, *Producer*, and *Recording*.

Figure 6.6 shows a description of a part of the structure of the information architecture. The nonterminal PROPS_MAIN_MODULE generates a textual description of the properties of the information architecture:

```
PROPS_MAIN_MODULE : PROPS_BAND,
                    PROPS_MUSICIAN,
                    PROPS_PERSON,
                    PROPS_RECORDING,
                    PROPS_SONG.
```

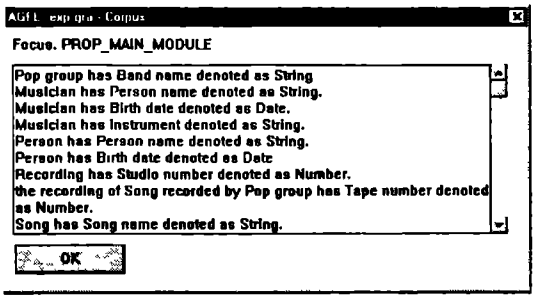


Figure 6.6: Paraphrasing properties

We assume bands to have a name, musicians to have an instrument, persons to have a name and birth date, recordings to have a studio number and tape number, and songs to have a name.

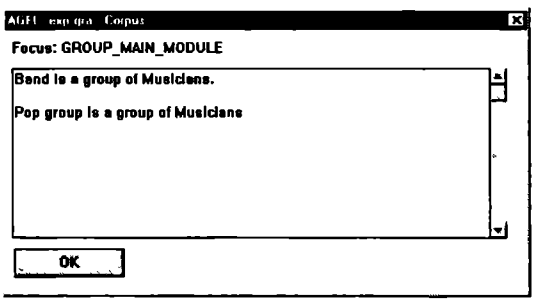


Figure 6.7: Paraphrasing structure

Finally, a part of the structure of the main module of the running example is paraphrased. Figure 6.7 focuses on possible groups of the main module. For this example a band is assumed to be a group of musicians. The nonterminal `GROUP_MAIN_MODULE` rewrites into:

```
GROUP_MAIN_MODULE : GROUP_BAND.
GROUP_BAND : BAND, "is a group of", MUSICIAN(plural), ".".
```

where `MUSICIAN(plural)` rewrites into the plural form of musician, i.e. *musicians*.

6.3.2 Parser generation

It is also possible to generate a parser for the information grammar using the parser generator GEN. Such a parser allows the domain expert to check whether sentences of the expert

language are captured by the information grammar. The sentence *Person joins Band.* is such a sentence of the expert language. Figure 6.8 shows the output of the parser of this sentence outlined by a parse tree. The lexical analysis, determining whether the words of a sentence are included in the grammar, took 0.002 seconds. The parsing of the sentence has been performed in 0.002 seconds, whereas the generation of the parse tree took 0.009 seconds.

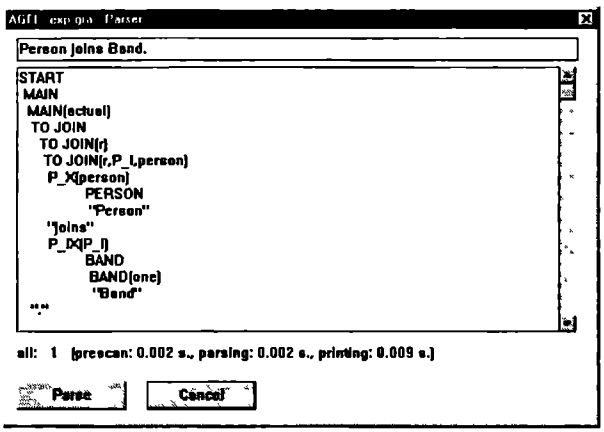


Figure 6.8: Parsing `Person joins Band.`

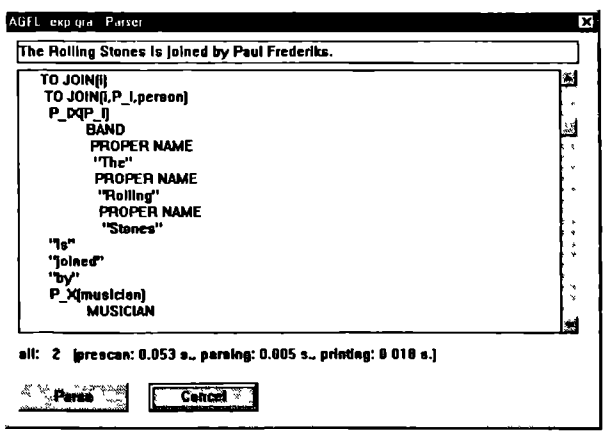


Figure 6.9: Dealing with proper names

A number of transformations are performed on the information grammar in order to accept random names for object types. For example the rule for MUSICIAN is modified as follows:

```
MUSICIAN: "Musician" ; PROPER NAME.

PROPER NAME: $MATCH("[A-Z][a-z]*"),[PROPER NAME].
```

where [PROPER NAME] denotes an optional member. Extending our grammar with these rules leads to a parser that allows sentences like:

```
The Rolling Stones is joined by Paul Frederiks.
```

Figure 6.9 shows a part of the resulting parse tree for this sentence. The string `The Rolling Stones` is recognized as a band. Note that the sentence:

```
Paul Frederiks is joined by The Rolling Stones.
```

would also be recognized by this parser, because we mainly consider syntactical aspects of natural language.

6.4 Existentional dependencies

In the object life model the course of life of object types is described via structural components and the task components in which they are involved. In this section the relation between the task components and their corresponding action types is further explored. We are especially interested in those cases where the birth of an object type (its initialization) comes about in an *interaction*, where an interaction is defined as an action type consisting of more than one predicator. In that case the object type is called *existentionally dependent*. This dependency can be detected by the so-called *life dependency graph* (cf. [DS95] or [Sno95]). In order to introduce this graph, we first consider the tasks from the object life model that start the life of an object instance.

From the predicate HasInit we find the initialization components of object types. An initialization component however, may be the root of a larger structure, in which case we are interested in the tasks that can be performed first. The predicate $\mathsf{IsStarter}(x,t)$ states that in the course of life of object type x task t may be performed first. Obviously, task t can only be reached via some initialization component via decomposition relations. Therefore we introduce the decomposition relation on components:

$$c\ \mathsf{Decomp}^*\ d \equiv c\ \mathsf{Connect}^*\ d \wedge \forall_{e,f}\,[c\ \mathsf{Connect}^*\ e \wedge e\ \mathsf{Connect}\ f \wedge f\ \mathsf{Connect}^*\ d \Rightarrow \mathsf{Decomp}(f) = e]$$

where $c\ \mathsf{Decomp}^*\ d$ states that component d can be reached from component c via decompositions only. The predicate IsStarter then can be defined as:

$$\mathsf{IsStarter}(x,t) \equiv \exists_c\,[x\ \mathsf{HasInit}\ c \wedge c\ \mathsf{Decomp}^*\ t] \wedge t \in \mathcal{T}$$

The life dependency graph can be seen as a special view on the object action involvement model, visualizing the special role of predicators with respect to existentional dependencies. Object types and action types are the nodes in this graph. Besides specialization and generalization edges, this graph has two more kinds of edges: *originator edges* and *descendant edges*. There is an originator edge from object type x to action type a if there is a predicator p such that:

$$\mathsf{IsOrigEdge}(x, p, a) \equiv \mathsf{Actor}(p) = x \wedge \mathsf{Action}(p) = a \wedge \chi(p) = r$$

There is a descendant edge from action type a to object type x if there is a predicator p such that:

$$\mathsf{IsDescEdge}(a, p, x) \equiv \mathsf{Actor}(p) = x \wedge \mathsf{Action}(p) = a \wedge \mathsf{IsStarter}(x, p)$$

The relation $\mathsf{IsAncestor}(x, y)$ describes whether there is a non-empty path in the life dependency graph from node x to node y.

Example 6.1

Figure 6.10 shows the life dependency graph for the running example. The bold dashed arrows are the originator edges whereas the bold arrows are the descendent edges. Note that an action type may have no incoming originator edge. For example, action type *to subscribe* is independent of other object types. Furthermore, action types may have no outgoing descendant edges. This is the case for action type that do not produce new object instances. An example is action type *to record*. One should pay attention to the fact that an action type may have outgoing originator edges. This happens when an action type is objectified.

The life dependency graph shows alle existence dependencies between object types. According to these dependencies the following classes for existentional dependence are distinguished:

1. *independent* object types. Object types that do not depend on other object types in any birth variant are called independent. Instances of these object types can only be created by subjects from the UoD. These object types are the nodes in the life dependency graph that have no incoming edges.

2. *dependent* object types. These object types correspond to nodes with an incoming edge. A dependent object type thus must have a birth task in which it is created by another object type. Object types which can (possibly via a number of intermediate steps) create instances of their own type are called *recursive*. In the type dependency graph, they correspond to nodes on a cycle.

Example 6.2

In figure 6.10 only action type *to subscribe* is an independent object type. All other object types are dependent. None of these dependent object types are recursive.

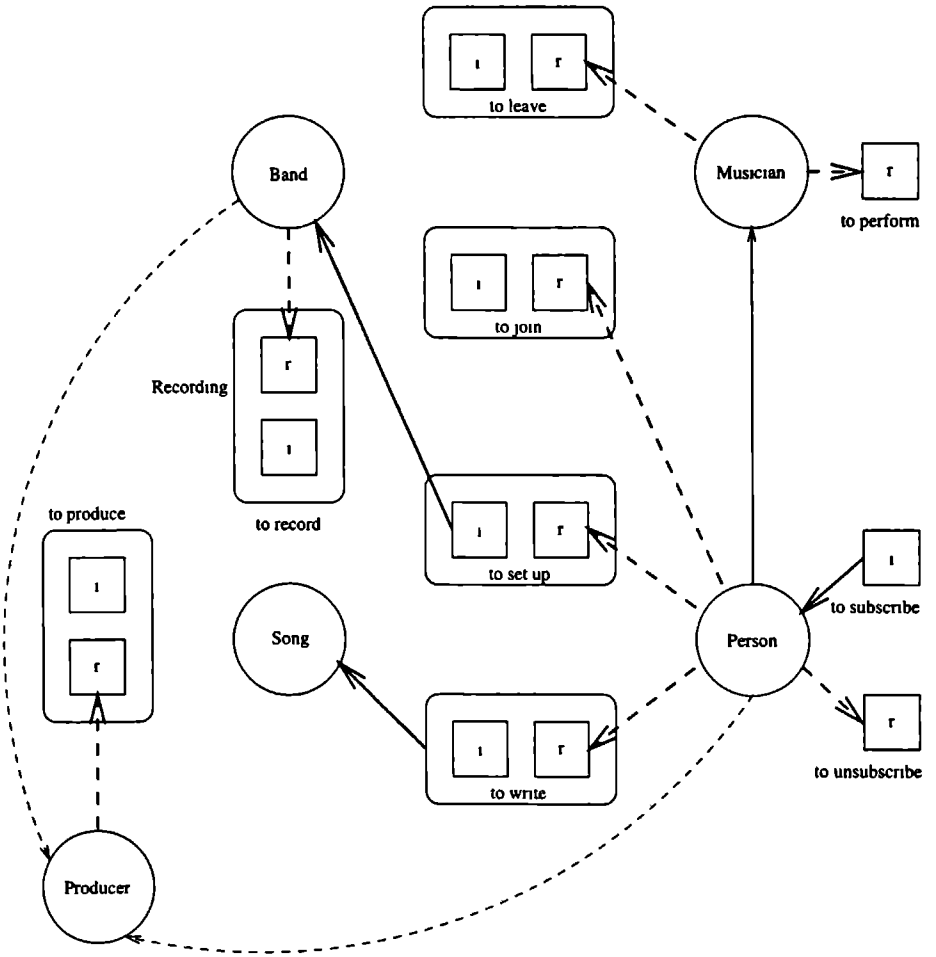


Figure 6.10: Life dependency graph of music example

Independent object types can be characterized by the following lemma:

Lemma 6.1 An object type x is independent if

$$\neg \mathsf{IsGen}(x) \wedge \neg \mathsf{IsSpec}(x)$$
$$\wedge \forall_p [\mathsf{Actor}(p) = x \Rightarrow \neg \mathsf{IsDescEdge}(\mathsf{Action}(p), p, x)]$$
$$\wedge \forall_p [\mathsf{Action}(p) = x \Rightarrow \chi(p) \neq r]$$

Proof:

Let x be an object type without incoming edges. Then obviously, x can not be a

specialized or generalized object type. Let p be predicator with x as actor. Then this predicator can not be a descendant edge. In the case that x is an action type, let p be a predicator of x. Then this predicator leads only to an incoming edge if $\chi(p) = r$.

An object type is recursive if there exists a non-empty path in the life dependency graph to itself. Recursive object types can formally be characterized by the following lemma:

Lemma 6.2 $\mathsf{Recursive}(x) \equiv \mathsf{IsAncestor}(x, x)$

Note that recursive object types require an independent birth variant for coming into existence at all (see example 6.3).

[LD-1] *(independent ancestor)* $\neg\,\mathsf{Independent}(x) \Rightarrow \exists_y\,[\mathsf{IsAncestor}(y, x) \wedge \mathsf{Independent}(y)]$

As a consequence, there is no a-priori restriction for object types to be instantiated.

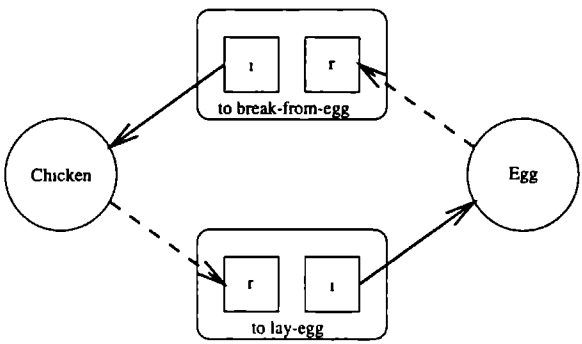


Figure 6.11: Chicken-egg problem

Example 6.3

A well-known example of a mutual life dependency is the *chicken-egg* problem, a typical example of recursive object types (see figure 6.11). As both object types *Chicken* and *Egg* are mutually dependent from each other, this magic cycle has to be broken before any of them can be instantiated. A typical solution is to add a special birth variant that initially enters some instance(s) into the system. From that point on, the involved object types will (must) be self-supporting. In the chicken-egg problem action type *to buy* is such a special birth variant for object type *Chicken* (see figure 6.12). Instances of object type *Chicken* are either bought or born from an egg. The advantage of this solution is that the system still may start in an empty initial state.

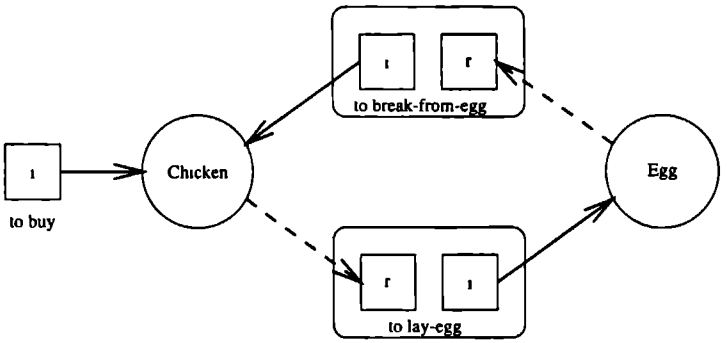


Figure 6.12: Chicken-egg life dependency graph

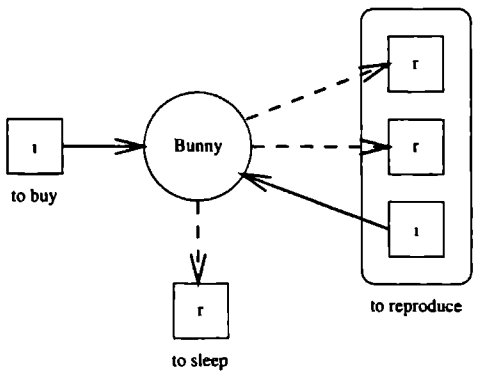


Figure 6.13: Life dependency graph for bunnies

Example 6.4

Suppose we have the following partial description of a life of an object type *Bunny*:

> *A bunny starts its life when it is bought or as a result of a reproduction activity. During its life time a bunny sleeps or is involved in reproduction.*

The life dependency graph is shown in figure 6.13. As we can see, object type *Bunny* is recursive.

Next we consider an instantiation L of the object action involvement model. This instantiation can be extended to a life dependency graph which describes the *family tree* of the instances.

Example 6.5

Figure 6.14 shows the family tree of instances of the chicken-egg problem. We consider

the following instantiation:

$$s_1 \in \mathsf{L}(\text{to buy})$$
$$c_1, c_2, c_3 \in \mathsf{L}(\text{Chicken})$$
$$x_1, x_2 \in \mathsf{L}(\text{to lay-egg})$$
$$y_1, y_2 \in \mathsf{L}(\text{to break-from-egg})$$
$$e_1, e_2 \in \mathsf{L}(\text{Egg})$$

The system starts with buying a chicken, recorded as s_1 (with $\pi_{\mathsf{i}}(s_1) = c_1$). This chicken c_1 lays an egg e_1 which gives birth to another chicken c_2, etc. In figure 6.15 we see the corresponding family tree for bunnies.

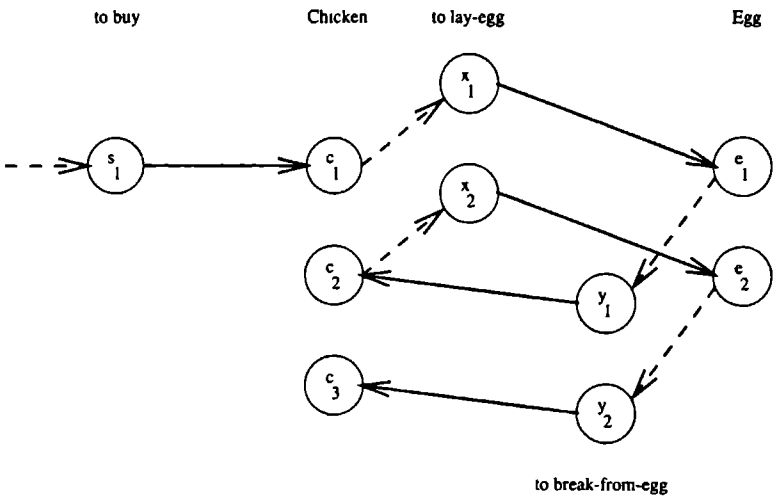


Figure 6.14: Family tree for chickens

A family tree may not contain a cycle as this would correspond to instances that are originated from themselves. Formally, let $\Omega_{\mathsf{L}} = \cup_{x \in \mathcal{O}} \mathsf{L}(x)$ be the set of all instances occurring in an instantiation L. Suppose $v, w \in \Omega_{\mathsf{L}}$. Then there will be an edge from instance v to instance w, denoted as $v \hookrightarrow w$, if for some object types x and y:

$$\mathsf{IsAncestor}(x, y) \wedge$$
$$v \in \mathsf{L}(x) \wedge w \in \mathsf{L}(y) \wedge$$
$$(\exists_p [\mathsf{IsOrigEdge}(x, p, y) \wedge \pi_{\mathsf{i}}(w)(p) = \pi_{\mathsf{i}}(v)] \vee \exists_p [\mathsf{IsDescEdge}(y, p, x) \wedge \pi_{\mathsf{i}}(v)(p) = \pi_{\mathsf{i}}(w)])$$

An instance v is called independent $\mathsf{Independent}(v)$ if it has no incoming edge. The acyclicity of family trees is enforced by the following schema of induction:

[LD-2] *(family induction)* Let Φ be a property for instances of Ω_{L} such that:

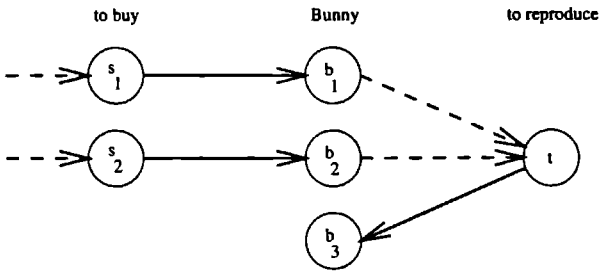


Figure 6.15: Family tree for bunnies

1. $v \in \Omega_{\mathsf{L}} \wedge \mathsf{Independent}(v) \Rightarrow \Phi(v)$
2. $\forall_{v \hookrightarrow w} [\Phi(v)] \Rightarrow \Phi(w)$

then we may conclude $\forall_v [v \in \Omega_{\mathsf{L}} \Rightarrow \Phi(v)]$.

With axiom LD-2 it is possible to sketch a proof for the following property:

Lemma 6.3 No bunny is an ancestor of itself.

Formally, this expression is included in the following predicate:

$$\forall_{b,v \in \mathsf{L}(\mathsf{Bunny})} \left[v \hookrightarrow b \Rightarrow \neg b \overset{*}{\hookrightarrow} v \right]$$

where $b \overset{*}{\hookrightarrow} v$ is an abbreviation for a path of zero or more edges between b and v.

Proof:

We introduce:

$$\Phi(b) = \forall_v \left[v \hookrightarrow b \Rightarrow \neg b \overset{*}{\hookrightarrow} v \right]$$

Using family induction we show $\forall_v [v \in \Omega_{\mathsf{L}} \Rightarrow \Phi(v)]$.

1. Suppose $\mathsf{Independent}(b)$. Then by definition b has no incoming edges and thus $\Phi(b)$.
2. Suppose as induction hypothesis $\forall_{c \hookrightarrow b} [\Phi(c)]$. Let $v \hookrightarrow b$. From the induction hypothesis we conclude $\Phi(v)$. Suppose there is a path from b to v. Let w be the predecessor of v on this path. The induction hypothesis denies a path from v to w, which is a contradiction with the assumption of a path from b to v.

6.5 Some design issues

In the design phase of an information system the object types that have been introduced in the analysis models are specified in more detail. The main goal of the design phase is to obtain an efficient representation of object types. This representation can be described in the style of traditional object-oriented design methods (see e.g. [CY90]). In these methods *object classes* are identified. Each object class has associated a number of *attributes* and a number of *methods*. Furthermore, object classes form *object class hierarchies*.

For each property, the designer has to find a balance between two extremes:

1. to introduce an attribute for this property. This allows for efficient retrieval of the value of the property at the cost of (1) memory space and (2) the obligation to keep the attribute up-to-date.

2. to represent the property as a method which computes on demand the value of the property from the current logbook.

For each action type it has to be decided whether (1) it is represented as a method of some object type, or (2) the action type is treated as an objectified action type. The life dependency graph gives a clue for this decision, as the originator edges suggest the associated action type to be a method of the corresponding actor.

An initial class hierarchy can be derived from the specialization and generalization relations of the object action involvement model. The object class hierarchy can be seen as an acyclic directed graph such that for each arrow from A to B it holds that object class A is a subclass of object class B. Object class A is a subclass of object class B if attributes and methods of object class B are a subset of the attributes and methods of object class A. Besides an initial class hierarchy more object class hierarchies are possible, by introducing classes which are not recognized at the conceptual level, i.e. classes which do not have necessarily a meaning in the UoD. The task for the designer is to find an optimal object class hierarchy.

In order to discriminate between good and bad internal models, it is necessary to estimate the time needed for actions (methods) operating on the information system data base and the required amount of the storage capacity (attributes). For this purpose, the notions of *access profile* and *data profile* have been introduced in relation with conceptual models (information architectures) in [Bom94]. The relative frequency of different data base operations is specified in an *access profile*. An access profile can be used to compare different internal representations with respect to their expected average response time. The expected size of the contents of the data base, i.e. the expected number of instances of object types, is recorded in the data profile. As usual time and space are conflicting objectives for optimization, which is reflected by the well-known *time/space trade-off*. For more reading on access profiles, data profiles and data base optimizations, the reader is referred to [Bom95].

6.6 Summary and outlook

This chapter has been discussing some validation, verification and design issues of the information architecture.

Validation of information grammars has been demonstrated using the AGFL formalism and a Window 95 user-interface for the AGFL system. The AGFL system is equipped with tools to generate and parse sample sentences describing events in the UoD. Especially, by generating a parser of the information grammar, the domain expert is actively involved in the validation of the information architecture.

Furthermore, it has been shown that there might not always exist an instantiation for a syntactically correct information architecture. Axioms have been introduced to reduce the number of information architectures which can not be instantiated. However, more research is necessary for methods to exclude such information architectures.

The design of information architectures has also been sketched. It has been made plausible that object classes with their attributes and methods can be obtained in a straightforward manner from the object-oriented analysis models. The definition of an object class hierarchy can be guided by an access profile and a data profile. More research for designing information architectures is necessary. A promising direction for this research is outlined in [KB96].

In the next chapter the semantics of the object action involvement model and the object property model is elaborated.

Chapter 7

Concepts and Constraints

Me I'm waiting so patiently lying on the floor
I'm just tryin' to do my jigsaw puzzle
Before it rains any more

From: "Jigsaw Puzzle",
The Rolling Stones

7.1 Introduction

In this chapter[1] a number of frequently used conceptual data modeling concepts are given a category theoretical foundation, resulting in a unifying framework. This framework should clarify the precise meaning of fundamental data modeling concepts and offer a sufficient level of abstraction to be able to concentrate on this meaning and avoid distractions of particular mathematical representations.

First, however, it is necessary to define a uniform syntax of conceptual data models that is as general as possible. In section 7.3, the syntax of conceptual data models is defined by means of *type graphs*. The semantics of a data model is the set of possible populations, i.e. instantiations of its structure. Populations are formalized via the notion of *type models*, defined in section 7.4. After the definition of type models, the various data modeling constructs are given a category theoretic definition (sections 7.5 to 7.8). These constructs are defined in terms of restrictions on type models. Valid type models and valid instance categories (which can be used to provide semantics to type graphs) are the subject of sections 7.9 and 7.10, respectively. Further restrictions on populations are represented by constraints. The most frequently used constraints (total role constraints and uniqueness constraints) are discussed from a category theoretical point of view in section 7.11. Finally, a summary of this chapter is presented in section 7.12. But first we start with a brief historical overview of category theory in section 7.2.

[1] This chapter is based on [HLF96] and [FHL97].

7.2 Background category theory

A brief history of the origin of *category theory* can be found in [McL92]:

> *Eilenberg and Mac Lane created categories in the 1940s as a way of relating systems of algebraic structures and systems of topological spaces in algebraic topology. The spread of applications led to a general theory, and what had been a tool for handling structures became more and more a means of defining them. Grothendieck and his students solved classical problems in geometry and number theory using new structures - including topoi - constructed from sets by categorical methods. In the 1960s, Lawvere began to give purely categorical definitions of new and old structures, and developed several styles of categorical foundations for mathematics. This led to new applications, notably in logic and computer science.*

Category theory therefore is a relatively young branch of mathematics designed to describe various *structural* concepts from different mathematical fields in a *uniform* way. Category theory offers a number of concepts, and theorems about those concepts, that form an abstraction of many concrete concepts in diverse branches of mathematics. As pointed out by Hoare ([Hoa89]):

> *Category theory is quite the most general and abstract branch of pure mathematics.*

In the seventies and eighties category theory has also found its way into computer science. Applications of category theory can be found in such diverse fields as automata and systems theory, formal specifications and abstract data types, type theory, domain theory, and constructive algorithmics. As pointed out by [Gog91], category theory can provide help with at least the following:

- *Formulating definitions and theories.* In computing science, it is often more difficult to formulate concepts and results than to give a proof. As stated by [AHS90], category theory provides a language with a convenient symbolism that allows for the visualization of quite complex facts by means of diagrams.
- *Carrying out proofs.* Once basic concepts have been correctly formulated in a categorical language, it often appears that proofs "just happen": at each step, there is a "natural" thing to try, and it works.
- *Discovering and exploiting relations with other fields.* Sufficiently abstract formulations can reveal surprising connections.
- *Formulating conjectures and research directions.* Connections with other fields can suggest new questions in one's own field.

- *Unification.* Computing science is very fragmented, with many different sub-disciplines having many different schools within them. Hence, the kind of conceptual unification that category theory can provide is badly needed.

- *Dealing with abstraction and representation independence.* In computing science, more abstract viewpoints are often more useful, because of the need to achieve independence from the overwhelmingly complex details of how things are represented or implemented.

This last item is particularly relevant in the context of this chapter. Category theory allows the study of the essence of certain concepts as it focuses on the *properties* of mathematical structures instead of on their *representation*. To illustrate this point, consider for example possible definitions of an *ordered pair*. The well-known Wiener-Kuratowski definition of an ordered pair is:

$$\langle a, b \rangle = \{a, \{a, b\}\}$$

From this definition one can always derive what the first element of the ordered pair involved was, and what its second element was. However, assuming that we deal with sets of natural numbers, the following definition also has this property:

$$\langle a, b \rangle = 2^a 3^b$$

Clearly, both definitions could be used for the definition of an ordered pair as both encompass its essence. However, it is also clear that they are both over-specific. One could speak of two *implementations* of ordered pairs. The definitions prescribe particular representations and do not focus on the underlying essence. They are precisely the kind of definition that category theorists abhor. One might say that category theory applies the Conceptualization Principle (as described in section 4.3) to mathematical formalizations.

Remark 7.1

Despite the popularity of category theory in some fields of computing science, not many applications in the field of information systems can be found in the literature. Recently, however, it seems that this is changing. Categorical formalizations of (aspects of) object-orientation (see e.g. [ES91], [FSMS91], [CSS94]), object-oriented data models (see e g. [Sie90], [Tui94], and [HH97]), ER (see e.g. [DJM92]), and the Relational Model (see e.g. [IP94], [BSW94]) have been proposed. In [SFMS89] a categorical framework for the axiomatization of conceptual modeling concepts is described (based on the notion of π-institution). In [Tui94] it is remarked that the uniformity of category theory provides a basis for interesting generalizations in the context of data modeling and that it not only offers insight in well-known operators but also allows for the definition of new operators, which would be far from trivial in other formalisms.

7.3 Type graphs

Data models can be represented by *type graphs* (cf. [Sie90] and [Tui94]). The various object types in the data model correspond to nodes in the graph, while the various constructions can be discerned by labeling the arrows. Relationship types, for example, correspond to nodes. An object type participating via a role (or predicator) in a relationship type is the target of an arrow labeled with role, which has as source that relationship type. As an object type may participate via several roles in a relationship type a type graph has to be a *multi*graph.

Definition 7.1

A *type graph* G is a directed multigraph over a label set $\{\text{role}, \text{spec}, \text{gen}, \text{elt-role}, \text{set-role}\}$. Edges with label spec or gen are called *subtype edges*. The type graph may not contain cycles consisting solely of subtype edges. Further, there is a bijective function Set_G from edges with label set-role to edges with label elt-role such that related edges have identical sources. The function EdgeType yields the label of an edge.

An edge e, labeled with role, from a node A to a node B indicates that A is a relationship type in which B plays a role. If e is labeled with spec, then A is a specialization of B, while if e is labeled with gen then B is a generalization of A (and possibly other object types). If edge $e : A \to B$ is labeled with set-role, edge $f : A \to C$ is labeled with elt-role, and $\text{Set}_G(e) = f$, then B is a set type with as element type C (set types will be explained in depth in section 7.7).

The definition of a type graph is very liberal, only cyclic subtype structures are (obviously) excluded. The definition allows a node to be a set type as well as a relationship type, a binary relationship type to be a subtype of a ternary relationship type, a set type to have several element types etc. Excluding these "peculiarities" from data models turns out to be unnecessary from a theoretical point of view as it is possible to give such data models a formal semantics. Hence, restrictions, other than on cyclic subtype structures, will not be imposed.

As an example of how data models can be represented as type graphs, consider figure 7.2, which shows the type graph of the NIAM data model in figure 7.1. Object types in NIAM are represented as circles, roles as boxes and arrows between circles represent subtype relations (for a complete overview of the graphical conventions of NIAM refer to [Hal95] and [NH89]).

7.4 Type models

The semantics of a data model is the set of all possible instantiations, also referred to as *populations*. In our approach, a population is defined as a *model* from the type graph to a category[2]. A model is a graph homomorphism from a graph to a category (interpreted as a graph).

[2]Appendix F contains the definition of the categorical constructs and notations needed in this thesis.

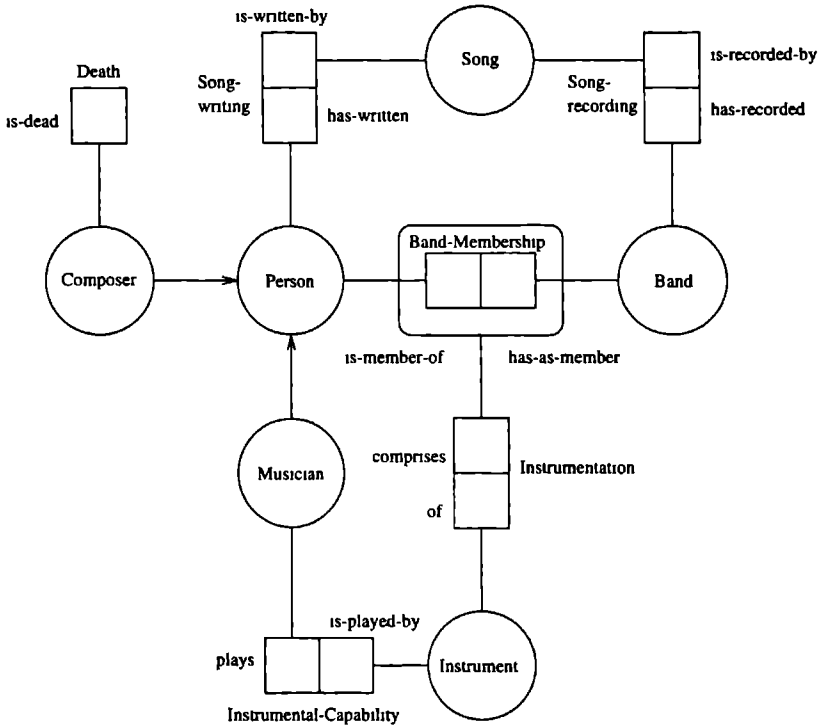


Figure 7.1: A NIAM data model

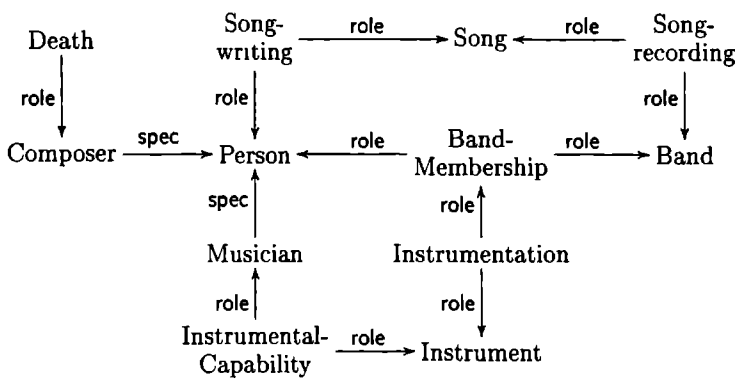


Figure 7.2: Type graph of the schema of figure 7.1

Definition 7.2

Given a category F, a *type model* for a given type graph G in F is a model $M : \mathsf{G} \rightarrow F$. F is referred to as the *instance category* of the model.

A type model maps the object types in the type graph onto objects in the instance category and the edges onto arrows in this category. To avoid notational clutter, the model is sometimes omitted if it is clear from the context. For example, the product of two object types is sometimes written as $A \times B$ instead of $M(A) \times M(B)$.

At this point no requirements on the mapping of edges in relation to their labels is imposed. These requirements will be discussed in the remainder of this section and will lead to the definition of a *valid* type model in section 7.9.

The above definition implies that the semantics of a data model depends on the instance category chosen. Not all categories provide a meaningful semantics for data models. Instance categories are required to be members of a class of categories **Fund**. Categories of this class have to fulfill a number of requirements that will be discussed in section 7.10.

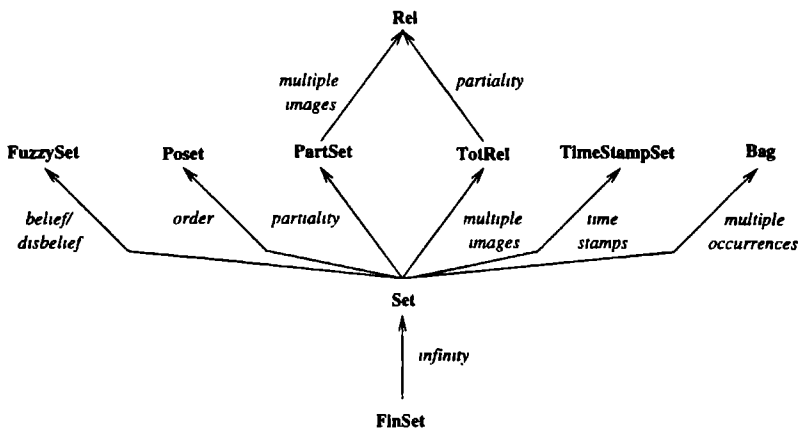


Figure 7.3: The class of categories **Fund**

In figure 7.3, some examples of categories in **Fund** are shown. The label of each arrow denotes a feature that exists in the category that is target of that arrow, but not in the category that is source of that arrow. For example, in the category **PartSet** functions do not have to be total, contrary to the category **Set**. As will be shown in section 7.5, this category should be considered if one is interested in the study of "null"-values in relationship types. Other categories in figure 7.3 are:

- The category **TotRel** where the objects are sets and the arrows total relations.

- The category **Bag** where the objects are bags (multisets) and the arrows total functions, such that the frequency of an original never exceeds the frequency of an image.
- The category **Poset** where the objects are partially ordered sets and the arrows monotonous (i.e. order-preserving) functions.
- The category **FuzzySet** where the objects are fuzzy sets and the arrows special total functions on these sets. A fuzzy set is a pair $\langle S, \sigma \rangle$ where S is a set and σ is a total function on S assigning to each element of S the degree of membership. An arrow $f : \langle S, \sigma \rangle \rightarrow \langle T, \tau \rangle$ is a function $f : S \rightarrow T$ such that $\sigma \leq \tau \circ f$.
- The category **TimeStampSet** where the objects are time-stamped sets and the arrows special total functions on these sets. A time-stamped set is a pair $\langle S, \tau \rangle$ where S is a set and τ is a total function on S assigning to each element of S a time stamp. The set of time stamps is a ordered set of discrete values. An arrow $f : \langle S, \tau_s \rangle \rightarrow \langle T, \tau_t \rangle$ is a function $f : S \rightarrow T$ such that $\tau_s \leq \tau_t \circ f$.

7.5 Relationship types

One of the central concepts in conceptual data modeling is the concept of *relationship type*. A relationship type represents an association between object types and may be n-ary in some data modeling techniques (where $n \geq 1$), as well as play a role in other relationship types. Yourdon ([You89]) refers to such relationship types as *associative object type indicators*, while in NIAM relationship types participating in other relationship types are called *objectified fact types*. A relationship type consists of a number of roles, capturing the way object types participate in that relationship type.

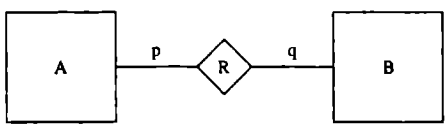


Figure 7.4: A simple ER schema

In the past, relationship types have often been formalized by viewing them as subsets of a cartesian product. This has commonly been referred to as the *tuple-oriented approach*. As an example consider figure 7.4 which depicts an ER schema ([Che76]) with a relationship type R consisting of roles p and q played by entity types A and B respectively. A population of this relationship type, represented in the tuple-oriented approach, could be:

$$\mathsf{Pop}(R) = \left\{ \langle a_1, b_1 \rangle, \langle a_2, b_1 \rangle \right\}.$$

The disadvantages of the tuple-oriented approach are obvious: the representation of instances is overly specific. Instances of relationship type R could as well be considered

elements of the product $\mathsf{Pop}(B) \times \mathsf{Pop}(A)$ as $\mathsf{Pop}(A) \times \mathsf{Pop}(B)$. A cartesian product imposes an ordering on the various parts of the relation. Consequently, the cartesian product does not have important properties such as commutativity and associativity. This observation has led to the *mapping-oriented approach* ([Mai88]), where relationship instances are treated as functions from the involved roles to values. In this approach, the above sample population would be represented as:

$$\mathsf{Pop}(R) = \Big\{\big\{p \mapsto a_1, q \mapsto b_1\big\}, \big\{p \mapsto a_2, q \mapsto b_1\big\}\Big\}.$$

Clearly, this approach does not suffer from the drawbacks of the tuple-oriented approach. No ordering is imposed, while at the same time the various parts of a relation remain distinguishable.

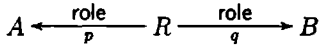


Figure 7.5: Type graph of figure 7.4

Still, however, one may argue that the mapping-oriented approach imposes unnecessary restrictions. Why do instances have to be represented as *functions*? Isn't it sufficient to have access to their various parts? The categorical approach pursues this line of thought. The actual representation of relationship instances becomes irrelevant, their components become available by "access-functions". As an example consider the interpretation of the sample population in the category **FinSet**. The type graph of the schema of figure 7.4 is shown in figure 7.5. Category theoretically, a population corresponds to a mapping from the type graph to an instance category. The sample population therefore, could be represented as (note that there are many alternatives!):

$$\begin{aligned} p &= \big\{r_1 \mapsto a_1, r_2 \mapsto a_2\big\}, \\ q &= \big\{r_1 \mapsto b_1, r_2 \mapsto b_1\big\}. \end{aligned}$$

In this approach, the two relationship instances, r_1 and r_2, have an identity of their own, and the functions p and q can be applied to retrieve the respective components. Note that in this approach it is possible that two different relationship instances consist of exactly the same components.

Apart from **FinSet** it is also possible to choose other instance categories. As remarked before, the category **PartSet** allows certain components of relationship instances to be undefined:

$$\begin{aligned} p &= \big\{r_2 \mapsto a_2\big\}, \\ q &= \big\{r_1 \mapsto b_1, r_2 \mapsto b_1\big\}. \end{aligned}$$

In this population, relationship instance r_1 does not have a corresponding object playing role p.

Another possible choice of instance category is the category **Rel**. In **Rel** the components of relationship instances correspond to sets, as roles are mapped on relations. A relationship instance may be related to one or more objects in one of its components. A sample population could be:

$$\begin{aligned} p &= \{r_2 \mapsto a_1, r_2 \mapsto a_2\}, \\ q &= \{r_1 \mapsto b_1, r_2 \mapsto b_1, r_2 \mapsto b_2\}. \end{aligned}$$

7.6 Subtype relationships

Many conceptual data modeling techniques offer concepts for expressing subtype relations. Subtype relations are used to capture inheritance of properties. In the literature many types of inheritance relations exist and the terminology is far from standard (see e.g. [Bra83]). In this section two important types of subtyping relations are considered: specialization and generalization. Many conceptual data modeling techniques contain at least one of these relations, although probably under a different name. The concepts of specialization and generalization in this chapter correspond to a large extent to specialization and generalization as defined in IFO ([AH87]).

7.6.1 Specialization

Specialization is used when specific facts are to be recorded for specific instances of an object type only. A specialized object type inherits the properties of its supertype(s), but may have additional properties. As such, specialization corresponds to the notion of *subtyping* in NIAM.

As an example of specialization consider the IFO schema of figure 7.6 (adapted from [AH87]). In this schema the boxes represent concrete types, the diamonds represent abstract types and the circles represent subtypes. The double arrows denote specialization relations. Therefore, in this diagram *STUDENT* is a subtype of *PERSON*. The object type *TEACHING-ASSISTANT* is a subtype of both *STUDENT* and *EMPLOYEE*. The subtype hierarchy has been created to express that only for certain types certain facts are to be recorded, e.g. only for employees the salary is relevant. As remarked before, properties are inherited "downward", e.g. employees have a name as they are also persons.

In set-theoretic terms, the most general formalization of a subtype relation would be to treat it as an injective function. This is more general than requiring that $\mathsf{Pop}(A) \subseteq \mathsf{Pop}(B)$ in the case that A is a subtype of B, as instances may have a different representation in both object types (this is particularly so in object-oriented data models). Therefore, category theoretically a subtype relation has to correspond with a monomorphism (recall that in the category **Set** a monomorphism corresponds to an injective function). This is not sufficient however for an adequate formalization of specialization relations. Consider for example the

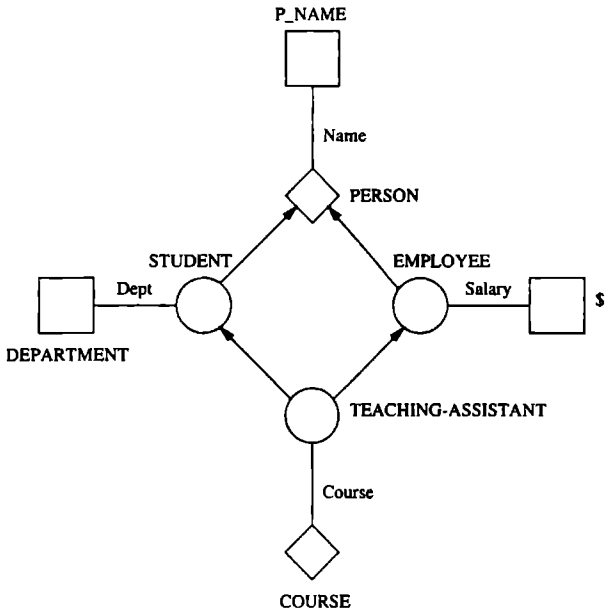


Figure 7.6: A subtype hierarchy in IFO

following partial population of the schema of figure 7.6:

$$\begin{aligned} \mathsf{Pop}(\text{PERSON}) &= \{\text{Jagger}, \text{Richards}\}, \\ \mathsf{Pop}(\text{STUDENT}) &= \{\text{ST1943}\}, \\ \mathsf{Pop}(\text{EMPLOYEE}) &= \{\text{EM237}\}, \\ \mathsf{Pop}(\text{TEACHING-ASSISTANT}) &= \{\text{TA999}\}. \end{aligned}$$

and the following subtype relations (see also figure 7.7):

$$\begin{aligned} I_1 &= \{\text{TA999} \mapsto \text{EM237}\}, \\ I_2 &= \{\text{TA999} \mapsto \text{ST1943}\}, \\ I_3 &= \{\text{EM237} \mapsto \text{Jagger}\}, \\ I_4 &= \{\text{ST1943} \mapsto \text{Richards}\}. \end{aligned}$$

In this sample population, with as instance category **Set**, the instance TA999 of object type *TEACHING-ASSISTANT* corresponds to two instances of *PERSON*: *Richards* as well as *Jagger*. Clearly, this is undesirable.

To avoid such problems, subtype diagrams, i.e. diagrams consisting solely of subtype edges, are required to commute. In terms of the presented subtype diagram this would imply that

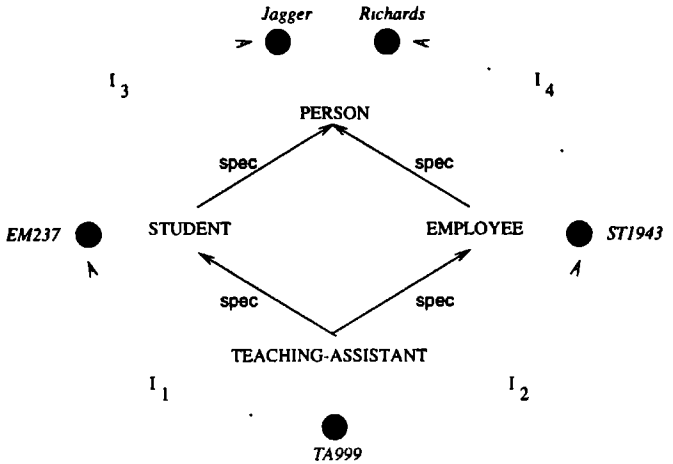


Figure 7.7: A non-commutative diagram

the function composition of I_2 with I_4 should be identical to the function composition of I_1 with I_3 and therefore: $I_4(I_2(\text{TA999})) = I_3(I_1(\text{TA999}))$.

Since the subtype diagram is required to commute, subtypes inherit properties from their supertypes in a unique way. In the example, every teaching assistant inherits the name from its supertype person.

7.6.2 Generalization

Generalization is a mechanism that allows for the creation of new object types by uniting existing object types. Contrary to what its name suggests, generalization is *not* the inverse of specialization. Specialization and generalization originate from different axioms in set theory ([HW93] or [HPW93]).

The population of a generalized object type is the union of the populations of the participating object types, referred to as the *specifiers*.

As an example of generalization consider figure 7.8. In this schema the graphical conventions of PSM ([HW93]) have been used, the dashed lines represent generalization relations. This PSM schema models the construction of simple formulas: a *Formula* may be either a *Variable* or an expression constructed by some function F from simpler formulas. This example demonstrates that generalization can be used for the specification of recursive types. Generalization is also useful when identical properties are relevant for different existing types: these properties can then be related to the generalization of these types.

The application of coproducts yields a possible categorical formalization of generalization. The generalized object type has to be mapped on a coproduct in the instance category and the generalization arrows should correspond to the sum injections. Of course, as the

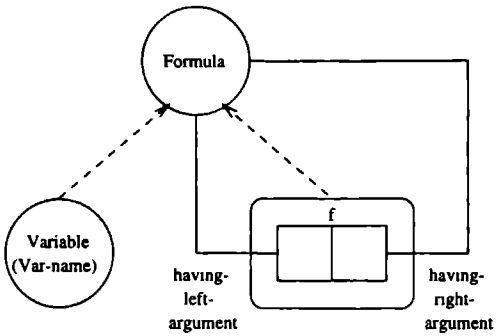


Figure 7.8: Generalization in PSM

coproduct represents a *disjoint* sum in **Set**, this formalization implies that specifiers have to be disjoint. In some data modeling techniques (including PSM) this is not necessarily true. This problem can be solved by using the general notion of colimit.

The solution starts with the observation that the collection of instances of a generalized type with a set of specifiers V is completely determined by the subtype relationships among the subtypes of elements in V. The following definitions give a formal description of a diagram that only contains the relevant subtype relations among subtypes of elements of V.

Definition 7.3

Given a graph G and a set of nodes $N \subseteq \mathsf{G}_0$, *the subgraph of* G *dominated by* N is equal to a subgraph D of G that is defined as follows: The edges of D are the edges from G_1 that occur on a directed path that ends in a node $n \in N$. The nodes of D are the nodes that occur in one of its edges.

Definition 7.4

Given a diagram $D : \mathsf{G} \to C$ and a set of nodes $V \subseteq \mathsf{G}_0$. Let G_V be the subgraph of G dominated by V. Then, D *dominated by* V is equal to D functionally restricted to G_V.

The instance universe U_M^V represents the collection of all instances of a set V of object types in a model M. The instance universe is used as the generalization of a set V of specifiers.

Definition 7.5

The *instance universe* determined by a set of object types $V \subseteq \mathsf{G}_0$ in a given type model M, denoted as U_M^V, is the apex of the universal cocone with as base the subtype diagram dominated by V.

In [LH96] it is proven that in a category that has disjoint sums the colimit of a diagram consisting of complementable monomorphisms always exists. This observation is true for

the subtype diagram of definition 7.5. The associated arrows are then also complementable monomorphisms. This result is important as some categories have disjoint sums, but do not have all colimits (e.g. **Rel**). Therefore, rather than requiring instance categories to have all colimits, it is required that all finite sums exist and are disjoint, as this is less restrictive.

Finally, it should be pointed out that as a result of the definition of subtype diagrams, the commutativity requirement imposed on these diagrams also applies to generalization.

7.7 Set types

A *set type* is an object type of which each instance corresponds to a (non-empty) set of instances of another object type. This latter object type is referred to as the *element type* of the set type. As sets are identical if and only if they contain the same elements, the instances of a set type are identified by their elements and do not need external identifications. Set types correspond to *grouping* in IFO, *association* in ECR ([EGH+92]), *grouping classes* in SDM ([HM81]), and *power types* in PSM.

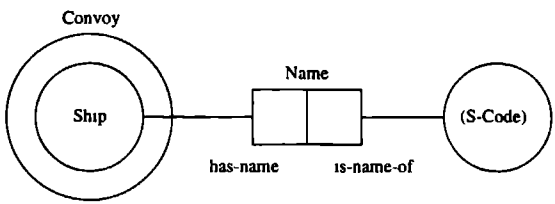


Figure 7.9: A set type in PSM

As a simple example of the application of set types consider the schema of figure 7.9, which shows a PSM schema of the so-called *Convoy Problem* of [HM81]. In this schema the object type *Convoy* is a set type with as element type *Ship*. Ships are identified by a code (*S-code*), while convoys are identified by their constituent ships.

There are several alternatives for a categorical formalization of set types. One alternative is to require the instance category to be a special kind of category called a topos. This approach has two serious disadvantages however. Firstly, a topos is a complex type of category, which is not easily understood. Secondly, and more seriously, many interesting categories are not topoi. The use of topoi therefore would imply an extra, very restrictive, requirement on the class of instance categories **Fund**. Another alternative would be the use of *sketches* in order to allow the general specification of algebraic types ([BW90b]). Unfortunately, it turns out that such a solution also imposes too many restrictions on **Fund**.

The approach adopted in this chapter does not suffer from the problems outlined in the previous paragraph and is based on an alternative treatment of set types, as presented

in [HW94]. As pointed out in this paper, set types become superfluous by the introduction of a new type of constraint, the *extensional uniqueness constraint*, as well as a new identification scheme. As an example consider figure 7.10. The extensional uniqueness constraint in this schema expresses that no two convoys may be associated, via role *sails in*, to the same set of ships. As such this constraint captures the extensionality property of sets. Also, the object type *Convoy*, may be identified, via this role, by the object type *Ship*.

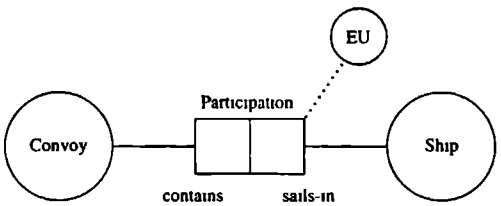
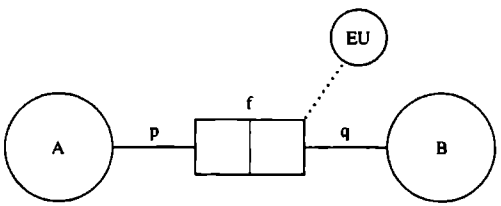


Figure 7.10: A translation of the Convoy Problem



f	$p(f)$	$q(f)$
f_1	a_1	b_1
f_2	a_1	b_2
f_3	a_2	b_1
f_4	a_2	b_2

Figure 7.11: A population violating the extensional uniqueness constraint

To further illustrate the extensional uniqueness constraint, consider the abstract schema of figure 7.11. The sample population of this schema violates the extensional uniqueness constraint as both a_1 and a_2 are related, via role q, to b_1 and b_2 and therefore both correspond to the set $\{b_1, b_2\}$.

The solution to the categorical formalization of the extensional uniqueness constraint follows from the observation that such a constraint is violated if and only if a non-trivial permutation of the "set-like" instances exists such that application to the population of

the involved relationship type yields *the same* population. In other words, if changing the members of two sets (which have their own identity!) does not lead to a loss of information, then obviously these two sets have to have identical representations. In the sample population the interchange of a_1 and a_2 in each instance of f, does not lead to a change in the population of relationship type f.

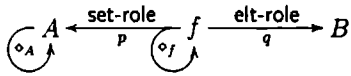


Figure 7.12: A solution for set types

Category theoretically, this requirement states that the extensional uniqueness constraint of the schema of figure 7.11 is violated if and only if the arrows p and q are mapped onto arrows in the instance category such that non-trivial isomorphisms (i.e. isomorphisms not equal to the identity) $\diamond_A$ and $\diamond_f$ on the objects, corresponding to the set type A and the involved relationship type f respectively, can be found for which the following equalities hold (see also figure 7.12):

$$\begin{aligned} \diamond_A \circ p \circ \diamond_f &= p, \\ q \circ \diamond_f &= q. \end{aligned}$$

The edges p and q are said to fulfill the *extensionality property*. Obviously, this definition does not impose any requirement on the instance category involved.

As an example of the application of this definition, again consider the sample population of figure 7.11. Suppose that the instance category involved is the category **Set**. The following two choices for the permutations $\diamond_A$ and $\diamond_f$ satisfy the imposed requirements, as they are non-trivial isomorphisms and satisfy the two equalities:

$$\begin{aligned} \diamond_A &= \{a_1 \mapsto a_2, a_2 \mapsto a_1\}, \\ \diamond_f &= \{f_1 \mapsto f_3, f_3 \mapsto f_1, f_2 \mapsto f_4, f_4 \mapsto f_2\}. \end{aligned}$$

7.8 Other complex types

Modeling techniques such as PSM and PSM2 also have so-called *sequence types*, and *schema types* or *modules types* (see section 4.2.1 and 4.3).

The left part of figure 7.13 shows a sequence type B with element type A. The relation type *IN* records the relation between the sequence type and element type. Via relation type *AT* it is recorded at which position an element occurs in a sequence. Label type I is the domain for indices in sequence types. The right part of figure 7.13 shows an alternative way to model a sequence type. This schema only contains regular object types and relation types which

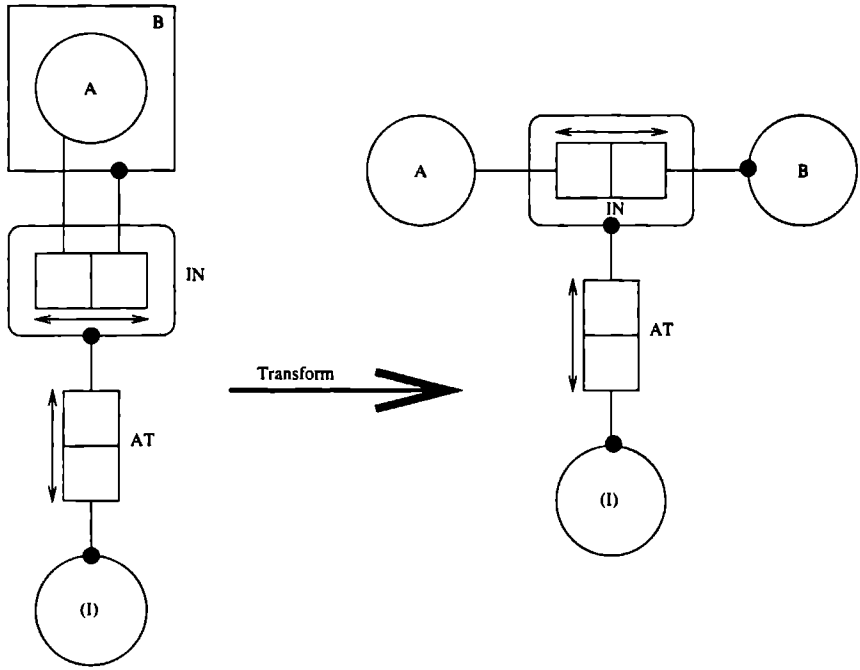


Figure 7.13: Transforming a sequence type

have been category theoretically dealt with in the previous sections. As a consequence the category theoretical framework as presented in this chapter is also applicable to sequence types. The constraints of this schema are discussed in section 7.11.

It is also possible to transform a schema type to a model using previously discussed concepts. The left part of figure 7.14 shows a schema type with underlying object types $X_1, \ldots, X_k$. Relation type IN_i records the relation between an underlying object type X_i and schema type S. The right part of figure 7.14 shows the model which results after transforming schema type S. The extensional uniqueness constraint as introduced in section 7.7 plays an important role in this transformation.

7.9 Valid type models

Now the full definition of a valid type model for a type graph can be presented:

Definition 7.6

A type model $M : \mathsf{G} \rightarrow F$ for a given type graph G in a category F, is a *valid type model* iff,

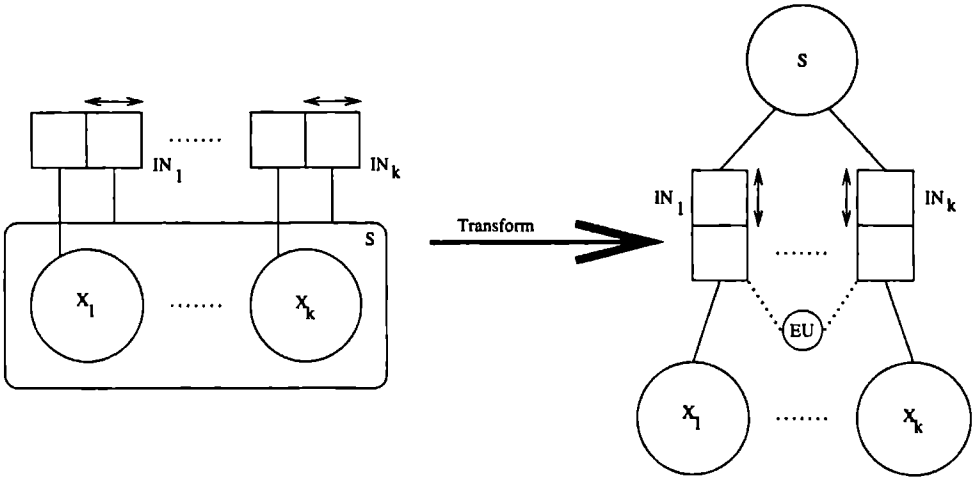
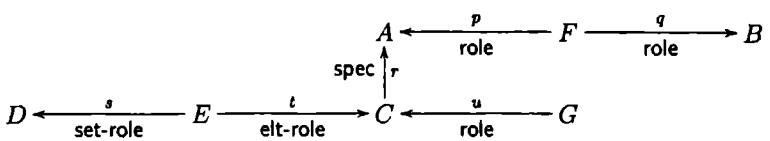


Figure 7.14: Transforming a schema type

1. if x is an edge of G and $\mathsf{EdgeType}(x) = \mathsf{spec}$ then $M(x)$ is a complementable monomorphism.
2. if x is an edge of G and $\mathsf{EdgeType}(x) = \mathsf{gen}$ then $M(x) = \alpha_D^{\mathsf{source}(x)}$, where D is equal to the subtype diagram dominated by the specifiers of $\mathsf{target}(x)$.
3. the subtype diagram of M commutes.
4. if x and y are edges of G, with $\mathsf{Set}_G(y) = x$ then $M(x)$ and $M(y)$ have to fulfill the extensionality property.

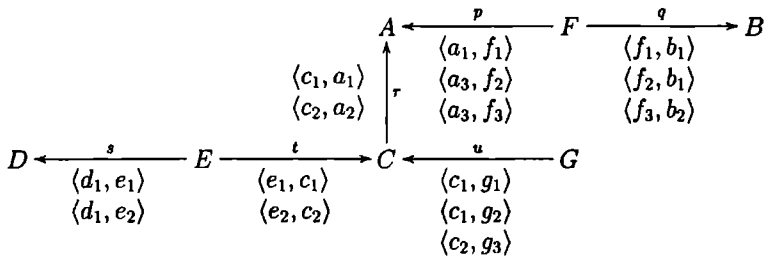
Example 7.1

The following type graph describes a simple conceptual data model.



The following is a type model of this type graph in **Set**. The value of the set of elements for each object is equal to the elements that occur in the corresponding

arrows, and has therefore been omitted from the figure.



This type model is indeed a valid type model. There is one specialization arrow from C to A that is an injective function, and in **Set** all injective functions are complementable monomorphisms. Obviously, the subtype diagram commutes since it only contains one specialization arrow. Set type D has one instance that represents the set $\{c_1, c_2\}$. It is not difficult to see that s and t fulfill the extensionality property.

7.10 Valid instance categories

One of the most important advantages of using a categorical approach to the semantics of conceptual data modeling techniques is that different instance categories can be used. The requirements that instance categories should satisfy are listed together with some illustrations.

Instance categories should support the constructions that have been used in the previous sections. This means that every member of **Fund** should have the following properties:

- All finite sums and products must exist.
- Sums must be disjoint.
- An initial object must exist.

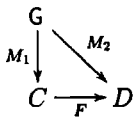
Actually, the last requirement is redundant since the initial object is the sum of zero objects. This set of requirements is modest, which implies that there is a large set of possible instance categories.

Some categories, however, are too trivial to be interesting as instance categories, for example the category with only one object and one arrow. Most "classical" formalizations of conceptual data modeling techniques correspond to a formalization that results from the choice of **FinSet** as instance category. Therefore, it seems reasonable to require that other instance categories have at least the same "expressive power". Intuitively, every model in **FinSet** should have a counterpart in other instance categories.

As an introduction to the formalization of this requirement it is useful to define a homomorphism between type models.

Definition 7.7

A *type model homomorphism* between type models $M_1 : \mathsf{G} \to C$ and $M_2 : \mathsf{G} \to D$ is a functor $F : C \to D$, i.e. a graph homomorphism preserving identities and composition, such that the following diagram commutes:



The valid type models and their homomorphisms form a category.

This definition of a type model homomorphism has inspired the following definition of a valid instance category.

Definition 7.8

A category C is a *valid instance category* iff all finite products and sums exist, sums are disjoint, and there is a functor $F : \mathbf{FinSet} \to C$ which is a monomorphism in the category of graphs and homomorphisms between graphs.

The categories **FinSet**, **Set**, **PartSet**, **Rel**, **TimeStampSet**, and **FuzzySet** are valid instance categories. A description of various category theory constructs and proofs for these categories can be found in [LH96] and [FW96a].

Example 7.2

In several object-oriented databases (see e.g. [KL89] or [ZM90]), objects can have multi-valued (or set-valued) attributes. This means that the value of an attribute can be a (possibly empty) set of attribute values. Models in the category **Rel** can be used to model this behavior.

Example 7.3

Models in **FuzzySet** can be used to model uncertainties. Every object type A is equipped with a function σ_A that captures the degree of membership of instances. For simplicity's sake, we assume that application of this function yields a probability (for an in-depth treatment of fuzzy sets in a categorical context refer to [BW90b]). The arrows in **FuzzySet** are total functions, and for each arrow $f : A \to B$ it must hold that $\sigma_A(a) \leq \sigma_B(f(a))$. Therefore, the probability that an individual is an element of a given object type must always be greater or equal to the probability that this individual is an element of one of the subtypes of this object type. Intuitively, this is sensible since if the individual is an element of an object type it must certainly be an element of all supertypes of that type. In addition to that, probabilities of instances of relationship types are less than the probabilities of their parts. If one considers for example the relationship type *Band-Membership* in the data model of

figure 7.1, one finds that the probability that a given person is member of a given band must be less than the probability that that person exists and also less than the probability that that band exists. So models in **FuzzySet** allow the introduction of uncertainty in conceptual data models in a natural way.

Example 7.4

Models in **TimeStampSet** are used to model the notion of time. This category is used in sections 5.2.4 and 5.3.2 to provide a semantics for the object action involvement model and object property model. Alternative ways to include time in the framework are discussed in [HLW97]. In that paper it is shown that it is possible to construct a new type model that describes type model changes over time for every valid instance category. A logbook as introduced in section 4.2 can be seen as a popular version of this category.

7.11 Constraints

Constraints represent restrictions on populations. They exclude populations that do not correspond with a possible situation in the problem domain. Consider for example the NIAM data model of figure 7.1. In this data model it may be desirable to express that each person is either a composer or a musician. This implies the specification of a constraint that enforces the populations of these object types to be a cover of the population of the object type person. In general, constraints may be quite complex and special languages for their specification exist (mostly founded in logic).

Two important *types* of constraints that are used frequently in conceptual data modeling techniques are the *total role constraint* and the *uniqueness constraint*. These constraint types correspond to a large extent to the cardinality constraints in ER. They are more general, as more than one relationship type may be involved. The semantics of these constraint types is described in the following sections.

7.11.1 Total role constraint

A total role constraint over a number of roles stipulates that all instances in the object types playing these roles have to participate in at least one of these roles. Total role constraints are important for applications as they determine mandatory/optional properties of objects. For example, in the relational model they determine whether a certain column is allowed to contain null-values.

Formally, a total role constraint in a given type graph G is determined by a set of edges $\tau \subseteq \mathsf{G}_1$. In the simplest example of a total role constraint, τ consists of a single edge e. This total role constraint means that all elements of $\mathsf{target}(e)$ must participate in e. In a model in the category **Set** this implies that $M(e)$ must be a surjective function. More generally we require that $M(e)$ must be an epimorphism.

Example 7.5

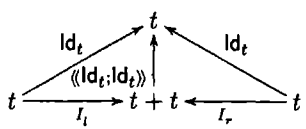
A total role constraint on the role with name *is-member-of* in the schema of figure 7.1, implies that every person has to be a member of a band.

A slightly more complicated example is $\tau = \{e_1, e_2\}$. Two cases can be distinguished, depending on whether both edges have the same target. In the first case both arrows have the same target $t = \mathsf{target}(e_1) = \mathsf{target}(e_2)$. The intuitive meaning of this constraint is that each element of t must participate in at least one of these two edges.

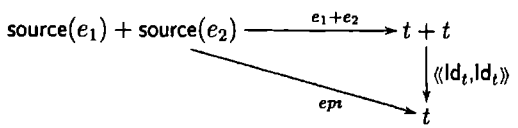
Example 7.6

In the context of the schema of figure 7.1, a total role constraint on the roles with names *is-member-of* and *has-written* implies that every person either is a member of a band or has written a song, or both.

For the semantics of this type of constraint, first construct the sum arrow $e_1 + e_2 : \mathsf{source}(e_1) + \mathsf{source}(e_2) \to t + t$. Intuitively speaking every element of t must be present in $\mathsf{target}(e_1 + e_2)$, however, as $t + t$ is a disjoint sum every element is represented twice. Therefore an arrow is needed that maps each element of $t + t$ onto the corresponding element of t. This can be achieved as follows. From the definition of the coproduct it follows that there are two injection arrows $I_l : t \to t + t$ and $I_r : t \to t + t$. Further, there is a unique arrow $\langle\!\langle \mathsf{Id}_t; \mathsf{Id}_t \rangle\!\rangle : t + t \to t$, such that the following diagram commutes.



The meaning of the total role constraint is that $\langle\!\langle \mathsf{Id}_t; \mathsf{Id}_t \rangle\!\rangle \circ (e_1 + e_2)$ must be an epimorphism.



If $\mathsf{target}(e_1) \neq \mathsf{target}(e_2)$, it is possible that one of these is a subtype of the other or e.g. that both types have a common supertype. In this case we first inject the elements of the subtype into the supertype and then follow the same procedure as in the previous case. Note that the supertype is always equal to $U_M^{\{\mathsf{target}(e_1), \mathsf{target}(e_2)\}}$.

Example 7.7

As an example of this type of total role constraint consider the schema of figure 7.15. A total role constraint on the roles with names *receives* and *earns-salary* would imply that every person, which is either a student or an employee or both, either owns a

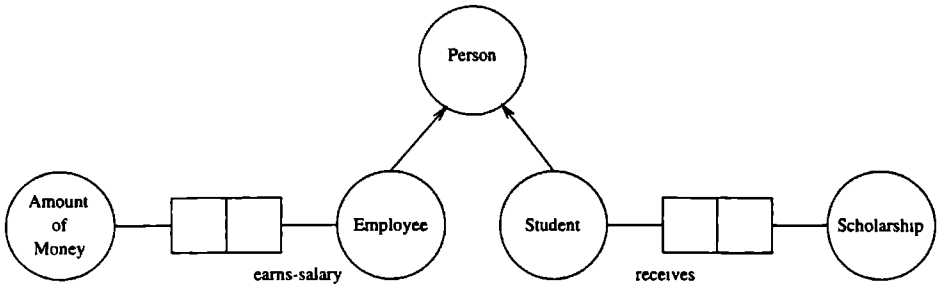


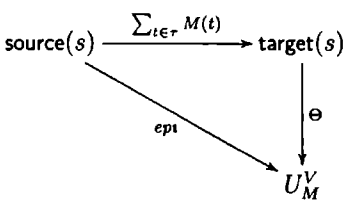
Figure 7.15: Sample schema

scholarship, or earns a salary. This is clearly different from the situation in which every student owns a scholarship and every employee earns a salary. As students may have different representations as employees, it is necessary to use the colimit construction to identify identical persons.

The full definition of the semantics of the total role constraint is given below.

Definition 7.9

Given a valid type model M and a total role constraint in the involved type graph G over $\tau \subseteq \mathsf{G}_1$. Let $s = \sum_{t\in\tau} M(t)$ and $V = \{\mathsf{target}(M(t)) | t \in \tau\}$. The definition of the instance universe U_M^V implies that for each $t \in \tau$ an arrow $i_t : \mathsf{target}(M(t)) \to U_M^V$ exists. Since $\mathsf{target}(s)$ is a coproduct, these i_t determine a unique arrow $\Theta : \mathsf{target}(s) \to U_M^V$. *$M$ satisfies the total role constraint τ* iff $\Theta \circ s$ is an epimorphism.



Example 7.8

In example 7.1 take the total role constraint over $\tau = \{p, u\}$. Then $V = \{A, C\}$ and $U_M^V = A$. The sum $p + u : F + G \to A + C$ is the function:

$$\{f_1 \mapsto a_1, f_2 \mapsto a_3, f_3 \mapsto a_3, g_1 \mapsto c_1, g_2 \mapsto c_1, g_3 \mapsto c_2\}$$

Then $\Theta : A + C \to A = \{a_1 \mapsto a_1, a_2 \mapsto a_2, a_3 \mapsto a_3, c_1 \mapsto a_1, c_2 \mapsto a_2\}$. The composition $\Theta \circ (p + u) = \{f_1 \mapsto a_1, f_2 \mapsto a_3, f_3 \mapsto a_3, g_1 \mapsto a_1, g_2 \mapsto a_1, g_3 \mapsto a_2\}$ is an

epimorphism in **Set** because it is a surjective function. Therefore, the total role constraint over $\tau = \{p, u\}$ is satisfied in this model.

The total role constraint over $\{p\}$ is not satisfied in this model (as a_2 is not in the range of function p), but the total role constraint over $\{q\}$ is.

The total role constraint can be seen as a generalization of several types of constraints found in conceptual data modeling techniques, such as the *collection cover constraint* and the *subtype cover constraint.* The collection cover constraint for a set type specifies that all instances of its element type should participate in at least one of its instances. The subtype cover constraint specifies that all instances of a given object type should be instances of at least one of a given set of subtypes of that object type.

7.11.2 Uniqueness constraint

The uniqueness constraint is closely related to the concept of a key over a relation. A uniqueness constraint in a given type graph G is determined by a set of edges $\tau \subseteq \mathsf{G}_1$.

In the most trivial case τ consists of a single edge e. The intuitive semantics is that each element of $\mathsf{target}(e)$ determines at most one element in $\mathsf{source}(e)$. For a model M in the category **Set** this implies that $M(e)$ must be an injective function. More generally, $M(e)$ must be a monomorphism.

Example 7.9

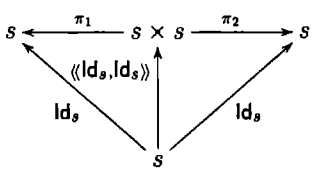
A uniqueness constraint on the role with name *is-written-by* in the schema of figure 7.1 implies that every song is written by at most one person.

In the next and more interesting case $\tau = \{e_1, e_2\}$ with $\mathsf{source}(e_1) = \mathsf{source}(e_2) = s$. In this case the intuitive semantics is that the combination of an element from $\mathsf{target}(e_1)$ with an element from $\mathsf{target}(e_2)$ determines at most one element in $\mathsf{source}(e_1)$.

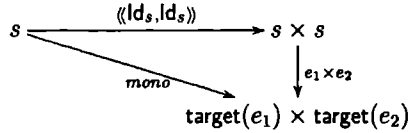
Example 7.10

Consider a ternary relationship between *Person*, *Duration*, and *Project*, capturing how many hours a certain person has worked for a certain project. A uniqueness constraint on the roles attached to the object types *Person* and *Project* expresses that a person-project combination has at most one associated duration.

Formally, start by constructing the product arrow $e_1 \times e_2 : s \times s \to \mathsf{target}(e_1) \times \mathsf{target}(e_2)$. From the definition of the product it follows that there are two projection arrows $\pi_1 : s \times s \to s$ and $\pi_2 : s \times s \to s$. Further, there is a unique arrow $\langle\!\langle \mathsf{Id}_s, \mathsf{Id}_s \rangle\!\rangle : s \to s \times s$, such that the following diagram commutes.



The meaning of the uniqueness constraint is that $(e_1 \times e_2) \circ \langle\!\langle \mathsf{Id}_s, \mathsf{Id}_s \rangle\!\rangle$ must be a monomorphism.

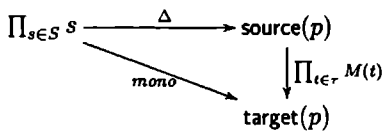


The case that $\tau = \{e_1, e_2\}$ with $\mathsf{source}(e_1) \neq \mathsf{source}(e_2)$ is simple, because it is equivalent to the combination of two uniqueness constraints over $\{e_1\}$ and $\{e_2\}$.

The full definition of the semantics of the uniqueness constraint is given below.

Definition 7.10

Given a valid type model M and a uniqueness constraint in the involved type graph G over $\tau \subseteq \mathsf{G}_1$. Let $p = \prod_{t \in \tau} M(t)$, $S = \{\mathsf{source}(M(t)) | t \in \tau\}$. For each $t \in \tau$ there is an arrow $\pi_t : \prod_{s \in S} s \to \mathsf{source}(M(t))$. From the definition of the product it follows that these π_t determine a unique arrow $\Delta : \prod_{s \in S} s \to \mathsf{source}(p)$. Then, M *satisfies the uniqueness constraint* τ iff $p \circ \Delta$ is a monomorphism.



As remarked in section 7.5, relationship types behave by default as multisets: the same tuple can occur more than once. If this is undesirable, it can be avoided by adding a uniqueness constraint over the roles of the relationship type.

Example 7.11

Take for instance, in example 7.1, the uniqueness constraint over $\tau = \{p, q\}$. Intuitively speaking, this constraint should be satisfied since every combination from A and B determines at most one element of F. The arrow $\Delta : F \to F \times F = \{f_1 \mapsto \langle f_1, f_1 \rangle, f_2 \mapsto \langle f_2, f_2 \rangle, f_3 \mapsto \langle f_3, f_3 \rangle\}$. The product $p \times q : F \times F \to A \times B =$

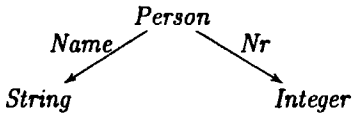
$$\begin{array}{l} \{\ \langle f_1, f_1 \rangle \mapsto \langle a_1, b_1 \rangle,\ \langle f_1, f_2 \rangle \mapsto \langle a_1, b_1 \rangle,\ \langle f_1, f_3 \rangle \mapsto \langle a_1, b_2 \rangle, \\ \quad \langle f_2, f_1 \rangle \mapsto \langle a_3, b_1 \rangle,\ \langle f_2, f_2 \rangle \mapsto \langle a_3, b_1 \rangle,\ \langle f_2, f_3 \rangle \mapsto \langle a_3, b_2 \rangle, \\ \quad \langle f_3, f_1 \rangle \mapsto \langle a_3, b_1 \rangle,\ \langle f_3, f_2 \rangle \mapsto \langle a_3, b_1 \rangle,\ \langle f_3, f_3 \rangle \mapsto \langle a_3, b_2 \rangle\ \} \end{array}$$

The composition $(p \times q) \circ \Delta = \{f_1 \mapsto \langle a_1, b_1 \rangle, f_2 \mapsto \langle a_3, b_1 \rangle, f_3 \mapsto \langle a_3, b_2 \rangle\}$ is a monomorphism, because it is an injective function. Therefore, the uniqueness constraint over $\tau = \{p, q\}$ is satisfied.

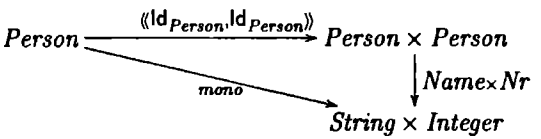
The separate uniqueness constraints over $\{p\}$ and $\{q\}$ are not satisfied because p and q are not monomorphisms.

Example 7.12

Models in **PartSet** give a way to handle missing values. Suppose that persons are identified by their names. Two different persons with identical names receive an additional number to distinguish them.



The arrow Nr is a partial function, because persons with a unique name do not have a number. Suppose that we want to express that every person must be uniquely identified by a combination of name and number. This can be achieved by putting a uniqueness constraint over $\{Name, Nr\}$.



The arrow $\langle\langle \mathsf{id}_{Person}, \mathsf{id}_{Person} \rangle\rangle = \{p \mapsto \langle p, p \rangle \mid p \in Person\}$. The arrow $Name \times Nr$ is interesting, since it maps the tuple $\langle p, p \rangle$ for a person p whose Nr is undefined to the tuple $\langle Name(p), \perp \rangle$. The uniqueness constraint holds if $(Name \times Nr) \circ \langle\langle \mathsf{id}_{Person}, \mathsf{id}_{Person} \rangle\rangle$ is a monomorphism, i.e. a total injective function. This implies that two persons with the same name must have different numbers, which was indeed the requirement we tried to express.

Some conceptual data modeling techniques, among others NIAM, allow uniqueness constraints over more than one relationship type. Such a uniqueness constraint expresses a key over a derived relationship type which is a join of the relationship types involved. Therefore, the semantics of this type of uniqueness constraint is completely determined by the way the join condition has to be computed (see also [WHB92]). As joins can be specified categorically by the use of pullbacks, we do not consider such uniqueness constraints explicitly. It should be remarked however, that some categories do not have pullbacks (e.g. **Rel**). In other words, the introduction of this type of uniqueness constraint leads to a further restriction on **Fund** (see also [LH96]).

7.12 Summary and outlook

This chapter presents a unifying framework for conceptual data modeling techniques. The framework is based on category theory due to its formality and its high level of abstraction. As has been pointed out, mathematical formalizations should not impose representational choices but instead focus on the *essence* of concepts. The framework described has been very general in the sense that 1) advanced conceptual data modeling concepts are incorporated,

2) very few syntactic restrictions on data models are imposed, and 3) the semantic target domain is not fixed.

The abstraction from representational issues allows the framework to be liberal with respect to syntax. The notion of instance category allows the framework to be liberal with respect to semantics. The framework offers opportunities for studying specific features in data models by offering a choice of corresponding categories as instance category. For example, the notion of time, which is used for the object action involvement model and the object property model can be studied via this framework.

The next chapter summarizes our research results and discusses directions for further research.

Chapter 8

Summary and Further Research

Spending too much time away
I can't stand another day
Maybe you think I've seen the world
But I'd rather see my girl

From: "Goin' Home",
The Rolling Stones

As stated in section 1.1 the focus of this thesis is on the *analysis phase* of the development of *information systems*. The success of an information system depends to a great extent on the *flexibility* of the information system in its ever changing environment, and on the way in which a user can *communicate* with the information system. Object-orientation and the explicit use of natural language for information system analysis seem to be good candidates for developing flexible and communication-oriented information systems, see section 1.2 and section 1.3, respectively.

An analysis method is required in order to perform information analysis. As stated in section 1.4 the goal of this thesis is to contribute a method for information analysis. This method provides a formal framework for the derivation, verification, and validation of object-oriented analysis models, using natural language as much as presently possible. These analysis models describe a so-called *information grammar* which governs the communication (the so-called *expert language*) in the *problem domain* or *Universe of Discourse* (UoD for short).

8.1 Summary

In chapter 2 the role of information grammars is elaborated. A discussion on the history and evolution of information system architectures and information grammars is provided in section 2.2. Furthermore, a general terminological framework ([Wij91]) for information methods, and thus for specifying information grammars, is discussed in section 2.3. This framework distinguishes:

1. a *way of thinking*,
2. a *way of controlling*,
3. a *way of working*,
4. a *way of modeling*,
5. a *way of validating*,
6. a *way of supporting*, and
7. a *way of visualization*.

This framework is the basis for the development framework described in this thesis.

Chapter 3 elaborates on the way of thinking, working, and controlling. The way of thinking of the framework is to derive and validate the analysis models using natural language. The pros and cons of natural language usage for conceptual modeling are presented in section 3.2. Obviously, a natural language based modeling process requires certain skills from those involved in the modeling process, i.e. the *system analyst* and *domain expert*. These skills, presented as *base axioms*, are also discussed in section 3.2. The base axioms are reflected in the way of working which is the subject of section 3.3. Furthermore, the products of the several stages of the development process are discussed in this section. In general, the modeling process is guided and controlled by a *project manager*. A number of managerial aspects with respect to object-oriented analysis methods in general, and natural language based methods are provided in section 3.4.

The main products of the development process are the subject of chapter 4. The notion of *logbooks* is introduced in section 4.2 as a unifying format for a natural language specification. This logbook can be seen as an extension of the format of natural language specifications in the methods NIAM ([NH89] or [Hal95]) and KISS ([Kri94]). From a logbook the so-called *information architecture* can be obtained. An information architecture is a conceptual model of the logbook and is composed of three object-oriented analysis models, being the *object action involvement model*, the *object property model*, and the *object life model*. The intuition behind the concepts, integration, and graphical notation (the way of modeling and visualization) of these analysis models is introduced in section 4.3. The resulting modeling technique is called PSM^2 and can be seen as an extension of the conceptual object-role modeling technique PSM ([HW93]).

A formal foundation for PSM^2 is described in chapter 5. In the sections 5.2 to 5.4 the syntax for each analysis model is described using first order predicate logic. The semantics of an object action involvement model and an object property model is defined using the category theoretical framework as described in chapter 7. The semantics of an object life model is described using process algebra, traces and histories. Each model describes a part of an information grammar of the UoD. For each model the relation with an information grammar is specified with pseudo code procedures.

Chapter 6 elaborates information grammars and information architectures. The AGFL formalism and system (see e.g. [Kos91]) as introduced in section 6.2 is used to implement a sample information grammar. The AGFL system is equipped with a grammar workbench and parser generator. The grammar workbench is used to produce sample sentences of the UoD. The parser generator can generate a parser from an information grammar. This parser allows system analysts and domain experts to check actively whether sentences of the expert language are captured by an information grammar. In section 6.3 the validation of an information grammar is demonstrated using a Window 95 user-interface for the AGFL system. Although an information architecture is syntactically correct, it is still possible that it is only partially populatable. In section 6.4 the so-called *life dependency graph*, which is a special view on an object action involvement model, is introduced to detect non-populatable models. Finally, in section 6.5 it is outlined how an object-oriented design for an information architecture can be obtained.

Finally, in chapter 7 a unifying category theoretical framework is presented which can be used to study modeling concepts and constraints. Since the framework contains most important concepts of existing data modeling techniques it can be seen as a generalization of these techniques. Therefore, the framework can be used to compare different conceptual data modeling techniques. First, however, it is necessary to define a uniform syntax for conceptual data models that is as general as possible. In section 7.3, the syntax of conceptual data models is defined by means of *type graphs*. The semantics of a conceptual data model is formalized via the notion of *type models*, defined in section 7.4. After the definition of type models, the various modeling concepts are given a category theoretic definition (sections 7.5 to 7.8). These concepts are defined in terms of restrictions on type models. Valid type models and valid instance categories (which can be used to provide configurable semantics to type graphs) are the subject of sections 7.9 and 7.10, respectively. Further restrictions on populations are represented by constraints. The most frequently used constraints (total role constraints and uniqueness constraints) are discussed from a category theoretical point of view in section 7.11.

8.2 Further research

Each chapter of this thesis contains one or more topics for future research. In the sequel of this section these topics are discussed for each chapter.

Chapter 2: The information system architectures and its components introduced in this chapter are globally outlined. In order to get a better understanding of what there has to be built, more detailed architectures seem to be appropriate. Other disciplines such as civil engineering have long-lasting experience with using detailed architectures for building their products. The discipline of computer science can and should learn from these other disciplines.

Chapter 3: Further research is required in order to investigate the consequences of the base axioms for the education of both domain experts and system analysts. The question of

whether the skills expressed by these base axioms can be learned seems to be justified. Furthermore, more research is necessary to get a better understanding of the cognition of domain experts and system analysts. It seems quite likely that the modeling process outlined in this chapter can be refined into smaller steps. An integration between the framework of this thesis (syntax-oriented) and other semantics-oriented methods (e.g. the COLOR-X method, see [Bur96]) can provide handles to get more grip on the modeling process. In the meantime more practical experience with the framework is required. Finally, the framework should provide more features with more details for a manager involved in the development process.

Chapter 4: Further research may address the completeness of the logbook: does the logbook contain al information relevant to the modeling process? Automatic transduction of informal specifications to logbooks is also a direction worthwhile studying. The expressiveness and the ease of use of the analysis models should be, and can only be, investigated in a real life case.

Chapter 5: For acceptance a modeling method has to have some automatic support. The formalization described in this chapter can be seen as a functional design for an implementation of the PSM2 modeling technique. Implementation and extension (see e.g. [Dal95]) of the paraphrasing mechanism described can add a surplus value to the framework. Cooperation with linguists is required to obtain better readable and more natural sentences.

Chapter 6: Validation of an information grammar can be simplified and made more attractive by implementing the Window 95 user-interface. In this thesis only paraphrasing of the structure of an information architecture is outlined. More research is necessary for paraphrasing constraints and populations of an information architecture. It is shown that an instantiation for a syntactically correct information architecture does not always exist. In [HW93] more handles for verifying conceptual models are provided. Designing information architectures is only shortly sketched. A few general heuristics for design are provided. More research for designing information architectures is necessary. A promising direction for this research is described in [KB96].

Chapter 7: Recently, more experience has been gained with the question whether application of the framework yields interesting (in)equivalence results ([HLW97]). In this latter paper several schema transformations, independent of any modeling technique, are category theoretically described. More study to the various members of the class of instance categories **Fund** seems to be worthwhile as well. An initial impetus to this study is presented in [LH96]. Currently, a prototype for the categorical framework is under development ([HBW95]).

Appendix A

Natural Language in Information Systems Engineering

In another land
where the breeze and the trees and the flowers grow blue
I stop and hold your hand
and the grass grew high and the feathers floated by

From: "In Another Land",
The Rolling Stones

A.1 Introduction

This appendix[1] discusses some aspects of natural language, and explains (where possible) the context in which natural language is applied in information systems engineering. We are aware of the fact that we embark on a slippery slope by discussing natural language properties in our role of computer scientists. Therefore, this appendix should be read as a handle (providing a number of commented references) for more readings about natural language properties.

A.1.1 Context

The application of *natural language* has become an important research area in computer science in general, and in information systems engineering in particular.

A first line to be mentioned focuses on information systems based on *structured information* (e.g. [Hal95]). These systems require information analysis, which is in many cases based on natural language (see also [NH89], [Win90], and [Kri94]). In addition, when validating an information model resulting from the analysis, natural language sentences can be generated (see e.g. [Dal95] and chapter 6). The latter is called *paraphrasing* of information models.

[1] This appendix is based on [VBFW96] and a chapter of [Ven96].

A second line focuses on information systems based on *unstructured information*, for example in the form of documents. This line is called Information Retrieval ([Rij75]), or IR for short. New techniques in Information Retrieval use document characterizations in which linguistic aspects are incorporated (see e.g. [Sme92]). The growing popularity of the World Wide Web (WWW) makes this even more urgent. In the information filtering project *Profile*, document characterizations are defined in terms of noun phrases ([HSB+96]). Also, the use of semantic relations such as synonyms, antonyms, hypernyms, meronyms, and homonyms is a promising direction here ([BB97]). However, despite several attempts to apply linguistics in Information Retrieval (e.g. [RS96]) have been made, a significant performance improvement is still to be demonstrated.

A.1.2 Overview

In [BH94] it is argued that a single sound definition of the term *language* covering all its necessary and sufficient aspects can not be given. Although linguistics has an empirical nature, language can only be defined as "that which is studied by linguistics". This definition of language clearly is unsatisfactory, but alternative definitions are often incorrect or incomplete, like the following attempt ([Gle91]):

language *the sounds produced by the vocal tract which have a meaning and are used to communicate.*

Note however that language is not always produced by the vocal tract (writing) and language is not always used to communicate (muttering).

When studying natural language a clear separation following Chomsky, between *competence* and *performance* of natural language has to be made. Competence is the knowledge a human being possesses about his mother tongue. This can be seen as a description of language as a system of *signs* and *meanings*. Performance on the other hand, is about *using* that competence by talking or writing natural language sentences. Although the knowledge about natural language is reflected in the use of it, language is sometimes used in such a way that it conflicts with our knowledge of how it *should* be used. For example, during a conversation we may restart, correct or even not complete our sentences.

The systematic part of language has been studied intensively by Noam Chomsky. Chomsky focuses on the systematic part, i.e. competence, of natural language which he uses as the basis of his *transformational-generative grammar* (*TGG*) ([Fah91]). The distinction between competence and performance is also mentioned by the Swedish linguist Ferdinand De Saussure, who stresses that the 'langue' (read competence or the language system) should be separated from the 'parole' (read performance or the way language is used). The 'langue' should be the subject of scientific analysis and should be studied by linguists. The 'parole', on the other hand, should be studied by psychologists and sociologists ([Mil91], [Fah91]).

In section A.2 five major *properties* of natural language are discussed. The properties *structure* and *meaning*, which are important for information systems engineering, are treated in sections A.3 and A.4, respectively.

A.2 Properties

In [Gle91] five major properties of natural language are distinguished. Language is:

1. *creative*,
2. *interpersonal*,
3. *structured*,
4. *meaningful*, and
5. *referential*.

The property of creativity states that humans are able to utter and understand sentences they have never heard before. Therefore, using language is not a kind of memorization that performs speech acts whenever the appropriate circumstances arise: language is a creative process.

The next property, inter-personality, states that using language is mostly a social activity in which the thoughts of one mind are conveyed to another. The other three properties mentioned will be discussed in more detail in the sequel of this section.

A.2.1 Language is structured

Although using language is a creative process, it is also restricted: there are unlimited numbers of strings of English words that we will *not* utter. As an example of an ungrammatical sentence consider:

The throws John ball

In information systems terminology the notion of *information structure* correspond with grammatically correct sentences (see e.g. [Hal95], [HW93]). The set of grammatically correct sentences may be restricted explicitly using *integrity constraints* ([HW93]) and an additional constraint language, such as Lisa-D ([HPW93]).

Our utterances are conform the abstract principles of the language we use. These so-called *structural principles* define how we can combine words into meaningful and comprehensible sentences. In general we are not aware of using these principles. Even so, these principles determine our use of language and allow us to compose and understand a boundless number of new sentences. This structural part of language is very systematic and therefore seems to be a candidate for formalization. In section A.3 more details and examples of the structure of natural language are provided.

A.2.2 Language is meaningful

Traditionally natural language can be divided into:

- *vocabulary*, consisting of all the words of a natural language.
- *grammar rules*, or *structural principles*, describing the rules by which sentences are constructed by combining the words from the vocabulary.

Although this division suggests a major emphasis on structure, usually semantic aspects are also incorporated here. This is for example the case in *lexica*, as used in [MBF+90], [BRC93]. In the area of Information Retrieval, lexica are used for obtaining advanced characterizations of documents (see e.g. [BB97], [Voo94]).

Each word in the vocabulary represents a meaningful *idea* (also called *concept*) about something (e.g. ball), action (e.g. to throw), abstraction (e.g. justice), quality (e.g. heavy), etc. In general, the relation between word and concept is defined arbitrarily. The purpose of language is to express all the meanings of utterances (references to concepts) to others, so we have no choice but to learn and memorize these relations.

But people talk in sentences rather than just one word at a time, and the structuring of words, i.e. composing sentences, form a major contribution to the meaning of our utterances. For example the words 'John', 'Paul' and 'killed', can be combined in two different sentences with very different semantics.

John killed Paul.
Paul killed John.

Note that the difference in semantics between the above two sentences is a result of different ways of structuring the same words. Handling semantics of sentences, will be discussed in more detail in section A.4.

A.2.3 Language is referential

Besides the fact that we know how to put words into meaningful sentences, we also know which words refer to which things, scenes and events in the real world around us (UoD).

Take as an example the sentence:

That's a rabbit.

The words are put together into a sentence in a correct and meaningful way. However, if this sentence is uttered by a little boy while pointing at a dog, we will not think he has learned English very effectively. This problem is called the problem of *reference*: how to use language to describe the world of real things and events. This problem is quite complex and belongs to the field of psychologists.

In information modeling, the term *reference* is often used in a slightly different way. A distinction is made between concrete (lexical) objects and abstract (non-lexical) objects, also called labels and entities, respectively. A relationship between a label and an entity is called a reference ([Hal95]) or bridge ([Win90]). A relationship between entities was originally called an *idea*, resulting in a Reference and IDea Language, RIDL for short ([Mee82], [DMP84]).

The study of the meaning and reference of words (and sentences) has been the subject of many classical philosophers. But the modern history of the philosophical discussion on meaning has been started by John Locke.

John Locke (1632-1704), an English philosopher, described in his *Essay Concerning Human Understanding*, published in 1690, that words are used to represent the ideas of the speaker: the meaning of a word is the idea that a speaker has in mind when he uses the word, and the idea the listener creates mentally when he hears the word. These ideas arise from perceptions. So to his opinion meaning and reference are each others equals. This theory can be successfully applied on words that refer to concrete objects in our real world. However, it does not hold for words that refer to *abstract* entities like 'justice' and 'conclusion'.

In the twentieth century the study of the meaning of natural language was greatly influenced by mathematical logic. In the article *Uber Sinn und Bedeutung*, published by Gottlob Frege (1848-1925) in 1892, the distinction between reference and meaning was made clear. Frege, a German mathematician and linguistic philosopher, separated the *reference* of a word (also called *denotation* or *extension*) from the *meaning*[2] of a word (also called *intention*).

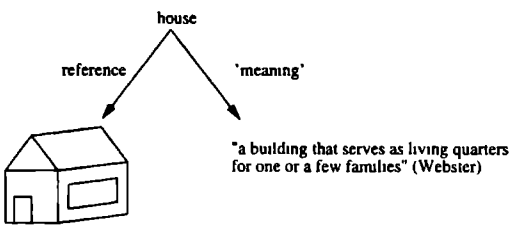


Figure A.1: Meaning separated from reference

Note that in this theory two expressions with different meanings can refer to the same reference-object. Take e.g. the following two expressions:

George Washington
the first president of the united states

Note that the problem of words referring to abstract entities is unlinked from the meaning but still remains unsolved in the view of some philosophers. The problem of reference is

[2]The meaning is given by means of a definition. In section A.4 other approaches to the problem of representation of meaning are presented. The problem of meaning is still a research issue.

an important issue in the field of psychology and philosophy but is beyond the scope of this section. Our attention will be focussed on the structure and the (mostly extensional) semantics of natural language.

A.3 Syntax

In this section the structure of natural language is investigated and different abstraction levels are distinguished. Sentences produced by *natural language users* are built by grouping phrases. These phrases in their turn are composed of words, and these words can also be split in smaller units called *morphemes*. A morpheme is a sequence of *phonemes*. This hierarchy is depicted in figure A.2 (adopted from [Gle91]).

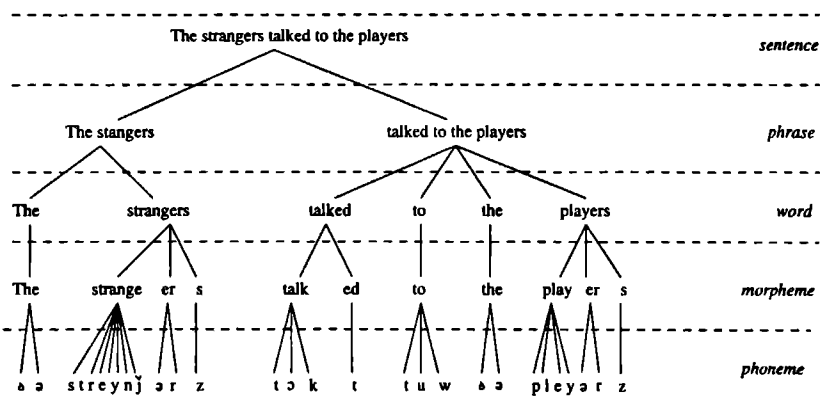


Figure A.2: Hierarchy of linguistic structures

Phonemes are described by symbols from the phonetic alphabet. The study of phonemes is called *phonology* and the study of morphemes *morphology*. The term *syntax* (i.e. 'arranging together' (Greek)) is the name of the system that arranges (or groups) words together into well-formed sentences. This topic has been investigated intensively by the American linguist Noam Chomsky and others. *Semantics* attributes a meaning to well-formed sentences (and, possibly also to some non-well-formed ones).

As stated before, the syntax of a natural language consists of two parts:

- *syntactical categories* i.e. the word families. All elements from a certain family can be exchanged with each other without changing the structure of the sentence.
- *syntactical rules* describing the possible arrangements of the syntactical categories.

Both syntactical categories and syntactical rules are discussed in the sequel of this section.

The structure of artificial languages has been studied intensively in the area of *programming languages* (see e.g. [ASU86]). Focus has been on the following three aspects: grammars, parsing techniques (e.g. [Ned94]), and compilers. Experiences gained with these studies have been applied to natural languages (see e.g. [KO96]). Of course information systems engineering has also benefited from the results gained here.

A.3.1 Syntactical categories

In indogermanic languages like English, traditionally ten syntactical categories are distinguished ([Mil91]): *noun*, *adjective*, *verb*, *adverb*, *pronoun*, *determiner*, *preposition*, *conjunction*, *numeral*, and *interjection*. These categories serve different, well-understood rôles in a sentence. In the stepwise procedure (way of working) of the conceptual data modeling technique NIAM ([Hal95]) the syntactical categories *noun* and *verb* play a major role. Nouns usually refer to object types, whereas verbs refer to roles played by object types in fact types. A similar approach to PSM ([HW93]) is presented in [CW93].

A category can be divided into sub-categories; these sub-categories can be divided into sub-sub-categories again. Take for example the category of nouns. Nouns can be divided into *proper nouns* (e.g. John, Yoko, The Beatles) and *common nouns*. The common nouns consist of *countable nouns* (book, table) and *non-countable nouns* (milk, grain, confidence).

Another example is the category of verbs. The verb-category can be split into three sub-categories:

1. *lexical verb* (to hear, to see, to register),
2. *auxiliary verb* (to have, to be),
3. and verbs expressing *modality* (can, must, may, shall, will).

The lexical verbs can be divided again into *intransitive*, *transitive*, *pseudo-transitive*, and *di-transitive* verbs. An intransitive verb (e.g. to sit, to sleep, to smile, and to talk) can not be combined with a direct object. For example we do not say[3]:

⋆(1) *John sleeps the bed.*
⋆(2) *Mary smiles the dog.*

Transitive verbs (e.g. to build, to adore, and to devour) on the other hand need a direct object. See the examples below:

(3) *John adores his mother-in-law.*
⋆(4) *Mary built.*

Sometimes verbs belong to both categories mentioned above. These verbs *can* have a direct object, but this is not necessary. This category of verbs is called the pseudo-transitive category and contains verbs like 'to eat', 'to drink' and to read'. This situation is shown by the examples given below:

[3] A ⋆ indicates the 'invalidity' of the sentence.

(5) *Peter drinks his milks.*
(6) *Peter drinks.*
(7) *Anne reads a book.*
(8) *Anne reads.*

The last category of verbs is called di-transitive and contains verbs like 'to sell' and 'to give'. The verbs in this category can have both a direct and indirect object.

(9) *John sells the car.*
(10) *John sells the car to Mary.*
(11) *Anne gives the book to Peter.*

A.3.2 Syntactical rules

A.3.2.1 Combining syntactical categories

After different categories for words have been distinguished, it needs to be studied how words and categories are related to each other. In conceptual data modeling techniques such as NIAM and KISS ([Kri94]), the relation between syntactic categories is represented via graphical models. These models represent so-called *structure sentences* or *elementary sentences*, which are a restricted form of natural language sentences.

Obviously, a category contains one or more words. On the other hand, in English, it occurs very often that a word belongs to more than one category (e.g. house, work, play, back, paper, surface, etc.). Some believe that such a word should be seen as a single word, while others say that we have to do with different words. The word 'house' as a noun has a phonetic representation that slightly differs from the word 'house' as a verb.

In written language however this difference can not be observed and may lead to confusions. Take for example the sentence below (adopted from [Gle91]). This sentence has two possible interpretations because both words ('bottle' and 'smell') belong to two different categories, namely noun and verb.

(12) *The French bottle smells.*

When 'bottle' comes out as a verb, the sentence is telling us something about what the French put into bottles - namely smells (i.e. perfumes). In another interpretation, the sentence is telling us something about some French bottle.

A sentence that can be interpreted in more than one way is called *ambiguous*. In the following lines a description of the syntactical rules is provided. With these rules ambiguity can be defined formally.

As mentioned before, syntactical rules describe how syntactical categories can be combined into sentences. Noam Chomsky has researched this intensively, and a lot of theories are based on his results ([BH94]). In [Cho65], Chomsky started the syntactical research, which resulted in a set of rules that describe how to combine the different syntactical categories. These rules can be expressed by so-called production rules of a context-free grammar ([ASU86]) with which the reader is supposed to be familiar.

A.3.2.2 Noun phrases and verb phrases

The combination of syntactical categories is important in Information Retrieval in the following context. The characterization of documents can be based on such combinations in order to overcome the simple description via keywords only. It has been proposed to use noun phrases as basic building blocks for document characterization, user profiles and queries ([HSB+96]).

As an example of combining syntactical categories, a *sentence* S can be seen as a sequence of a *noun phrase* NP and a *verb phrase* VP.

(SR1) ⟨S⟩ : ⟨NP⟩ ⟨VP⟩

A noun phrase consists of a noun N, that can be preceded by a *determiner* Det and followed by a *prepositional phrase* PP. A noun can be preceded by an unlimited number of *adjectives* Adj. A prepositional phrase is a *preposition* P followed by a noun phrase. In the following grammar, optional arguments are denoted between brackets.

(SR2) ⟨NP⟩ : [Det] ⟨N⟩ [⟨PP⟩]
(SR3) ⟨N⟩ : [Adj] ⟨N⟩ | N
(SR4) ⟨PP⟩ : P ⟨NP⟩

Because of the different syntactical sub-categories of a verb V, a verb phrase can be constructed in different ways. The intransitive verb has no (in)direct object, the di-transitive can have both a direct and indirect object. The following rule provides the necessary syntactical rules for the intransitive, transitive and di-transitive verbs.

(SR5) ⟨VP⟩ : V | V ⟨NP⟩ | V ⟨NP⟩ ⟨PP⟩

With these syntactical rules sample sentences can be parsed and generated automatically (see e.g. [DKNZ92b]). The above mentioned rules are also called rewrite rules.

A sentence can be represented by a *parse tree* in which each internal node represents the application of one of the rewrite rules. In figure A.3 the parse tree of the sentence:

The neighbor of my brother sells the car to Mary

is depicted. This example is adopted from [BH94]. Ambiguity can now be defined as follows:

A sentence is ambiguous if it has more than one parse tree.

Figure A.4 shows that sentence (12) is indeed ambiguous by this definition.

A.3.3 Transformation rules

Although these rewrite rules are a powerful mechanism to analyze and describe certain sentences from our language, they are not powerful enough to describe other sentences like (13b) and (14b)[4].

[4]adopted from [BH94]

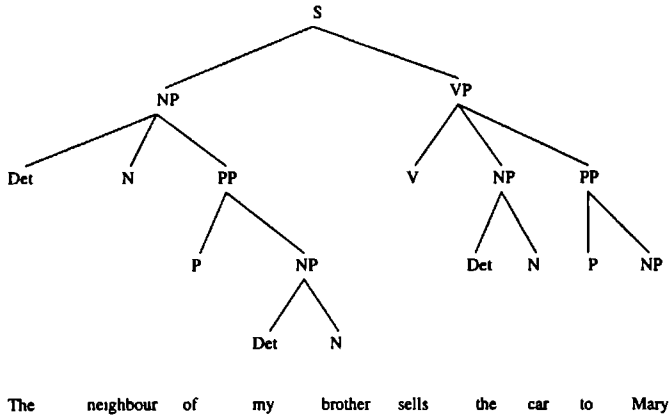


Figure A.3: Parse tree of a sample sentence

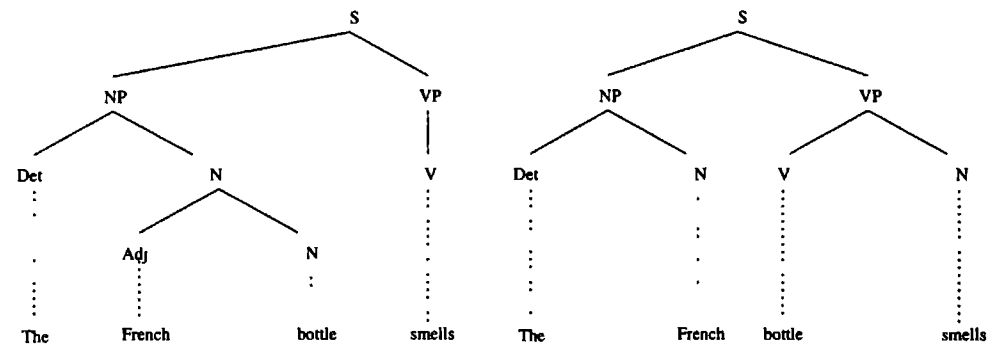


Figure A.4: Parse trees of an ambiguous sentence

(13a) *Peter read the book.*
(13b) *Did Peter read the book?*
(14a) *The barbarians destroyed Rome.*
(14b) *Rome was destroyed by the barbarians.*

Chomsky chose not to extend the rewrite rules with rules to produce interrogative (13b) and passive (14b) sentences, but to introduce another type of rule: the so-called *transformation rules*. It is clear that in contrast with (13b) and (14b), sentence (13a) and (14a) can be produced by the above mentioned set of rewrite rules. A suitable transformation rule can now be applied to (13a) to produce (13b). This rule transforms a declarative sentence into an interrogative sentence. Another transformation rule may be used to transform the active sentence (14a) to the passive sentence (14b). So, to recapitulate, the sentences (13b) and (14b) may be produced indirectly, by first applying the rewrite rules, producing a so-called *deep structure*, and secondly by the transformation rules, producing the so-called *surface*

structure ([BH94]). This situation is depicted in figure A.5, which is adopted from [BH94].

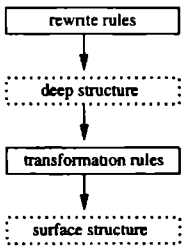


Figure A.5: Model of the transformational - generative grammar

A problem with transformation rules is that (in principle) these rules can be defined arbitrarily. A transformation rule can transform everything into everything. Restrictions are needed to distinguish meaningful transformation rules from senseless transformation rules. The restrictions on these rules can only be obtained by induction on an empirical basis. The last 25 years of research on syntactical aspects of language has been focussed mainly on finding empirical restrictions and on formulating these restrictions adequately.

Distinguishing rewrite and transformation rules facilitate the description of language, and to check sentences on well-formedness on a syntactical level. In the following section it is described how semantical anomalies, i.e. sentences without an explicable meaning (e.g. *The table writes a letter*), can be excluded.

In the field of computer science a lot of experience has been gained with transformations on 'sentences' of programming languages ([Par90]). Transformations are also important in the field of compiler construction, where they are often described by *transduction.* The idea behind transduction is to transform 'sentences' in one language to 'sentences' in another language. In this context, language should be considered in a wider sense. For example, in information systems engineering, structured sentences are transduced to information models (NIAM and KISS).

A.4 Semantics

In this section, based on [Mil91], it is described how the meaning of our utterances can be defined and used. A discussion on the meaning of single words is provided, followed by a discussion on meaning of complete sentences.

A.4.1 Meaning of words

It is generally accepted by linguists that the meaning of a word is a *concept*. So a word is a representative of a meaningful idea (or concept) in the mind of the person using that word. During communication we try to express these concepts to others. The question,

How to express these concepts?

is one of the most difficult questions in the study of language. The subfield that attempts to answer this question is called the study of *semantics*.

Currently words and relations between their meaning are extensively studied, resulting e.g. in lexica with some form of *ontology*, such as WordNet [5], which can be used for several directions in informations system engineering research (see e.g. [BR96b]). The ontology provides the user with the possibility to search for e.g. synonyms, antonyms, hypernyms, and homonyms.

As stated before, one of the oldest approaches to the topic of meaning equates concepts of words and phrases with reference. We have already seen that this theory of meaning results in several difficulties. Therefore two other approaches are defined: the *definitional theory of meaning* and the *prototype theory of meaning*.

A.4.1.1 Definitional theory of meaning

In figure A.1 the word *meaning* has been placed between quotation marks, because - as just stated - the meaning of a word is a concept rather than a definition. A concept is an idea in someone's mind, a definition however is written by the author of a lexicon. It is the *definitional theory* of meaning that assumes that these two fulfill the same role.

This approach states that meaning can be analyzed into a set of subcomponents, organized in our minds as they are in standard dictionaries. Various meaning (or semantical) relationships exist between words and phrases. Some words are similar in meaning (*synonym* relationship) and other are opposites (*antonym* relationship). According to this approach these relations can be explained by assuming that words are sets of *semantic features*. The concept of 'bird' for example contains the semantic features 'feathers', 'flies', 'animal' and 'wings'.

Taken together, the semantic features constitute a definition of a word. According, to this theory, we carry such definitions in our heads as the meaning of words in a so-called *mental lexicon* ([Mil91]).

Although this approach is intuitively satisfying, some problems arise: language itself is used to express the meaning of language-elements. To break this vicious circle, philosophers - from Plato to Leibniz - have been trying to make a list of so called axiomatic 'base-words', that can be used to describe the meaning of all other words ([Mil91]). Up to now all these attempts have failed.

[5] The WordNet home page is accessible via the internet: http://www cogsci princeton.edu/~wn.

A.4.1.2 Prototype theory of meaning

Another approach that attempts to define concepts is the *prototype theory* of meaning. This theory explains the fact that some members of a meaning category appear to exemplify that category better than others do. The definitional theory, lists the necessary and sufficient semantic features that *define* a concept. However an 'armchair' seems to be a better example of the concept of 'furniture' than a 'reading lamp'. An armchair is a typical piece of furniture, a reading lamp is not. This difference cannot be explained by the definitional theory of meaning. It claims to have said it all by the following definition of 'furniture' (conform Webster's dictionary).

furniture *movable articles used in making a room ready for occupancy or use*

The prototypical theory states that a concept is defined by a whole set of features, no one of which is individually either necessary or sufficient. For example, consider the concept of 'bird'. Prototype theorists claim that 'birdiness' is determined by the total number of features a given creature exhibits. Animals that have few (penguins and ostriches) will be judged to be poor members of the bird family, while those that have many (such as robins) will seem to be a strong member (or prototype).

It appears that both the definitional and the prototypical approaches to word meaning have something to offer. The prototype theory explains why a robin is a 'better' bird than an 'ostrich', whereas the definitional theory says that an ostrich should nevertheless be classified as a bird (conform Webster's dictionary).

bird *any of a class of warm-blooded vertebrates distinguished by having the body more or less completely covered with feathers and the forelimbs modified as wings*

A.4.2 Meaning of sentences

So far, it has been discussed how to handle the meaning of words. When studying the meaning of sentences we come to a problem at a higher level of complexity. It has been argued that the meaning of a single word is a concept which can be 'described' by means of a definition or a prototype. The meaning of a complete sentence on the other hand, has to do with relations *between* these concepts. Evidently, the context-sensitive nature of natural language - ignored by the rewrite-rules of the transformational-generative grammar - supplies a major contribution to the meaning (i.e. semantics) of natural language.

Basic sentences introduce some concept that they are about (called the *subject* of the sentence). Simon C. Dik mentions ([Dik89]) that the term 'subject' only makes sense when a certain context (i.e. the *predicate* of the sentence) is provided. In a sentence like (15) 'The boy' is called the subject, and what is proposed or predicated of this concept is that he 'hit the ball'.

(15) *The boy hit the ball.*

Generalizing from this example, the meaning of a sentence (sometimes called *proposition*) can be regarded as a sort of miniature drama in which the verb is the action and the nouns are the performers, each playing a different role. In the example above of the *boy-hitting-ball* miniature drama, the 'boy' is the *do-er*, 'the ball' the *done-to* and 'hit' is the *action* itself ([Gle91]). This *action-oriented* viewpoint treats verbs as basic frames to be filled with concepts. This approach enables us to describe the meaning of certain sentences precisely.

However, the linguistic theory of the transformational-generative grammar is a formal attempt at structuring syntactical categories in terms of rules of formal syntax to be applied *independently* of the meanings. In this theory, syntax is thus given priority over semantics. Of course, semantics is recognized by this theory, but is not its basic assumption. How this linguistic theory handles semantics will be described shortly.

lexical item	to smile	to eat	to sell
category	V	V	V
sub-categorization	<>	$< (NP) >$	$< NP(PP) >$

Table A.1: Lexicon containing context-sensitive information

The transformational-generative theory states that context-sensitive information should be stored, separately from the rewrite-rules, in a lexicon. This implies that the context-sensitive information describing to which sub-category a verb belongs, should also be stored in the lexicon (see table A.1).

Note that this context-sensitive information is needed for the benefit of a correct deep-structure generation. Therefore the lexicon is consulted *during* deep-structure generation by choosing the appropriate lexical items to fill the structure. With this information semantic anomalous sentences like,

John walks the house,

can be excluded.

Another form of context-sensitive information are the so-called *selection restrictions*. The selection restriction expressed in table A.2, states that the PP of 'to talk' should be animate. With this restriction rule, semantic anomalous sentences like,

John talks to the table,

can be excluded.

Lately, lexica are used in modeling methods and case tools for the development of information systems supporting both system analyst and domain expert. Example are OICSI ([RP92]), LOLITA ([MG94]), and COLOR-X ([Bur96]).

lexical item	to talk
category	V
sub-categorization	$< (PP_{[animate]}) >$

Table A.2: Example of selection restriction

Appendix B

Sample Logbook Population

Fingerprint file, you get me down
You get me running, keep me on the ground
Know my moves, way ahead of time
Listening to me, on your satellite

From: "Fingerprint File",
The Rolling Stones

Time	Event			
	Action	Associate		
		Involved	Role	Properties
01-05-1963	set up	Mick Jagger	agent	$\varnothing$
		Keith Richards	agent	$\varnothing$
		The Rolling Stones	object	$\varnothing$
03-05-1967	write	Mick Jagger	agent	$\varnothing$
		Keith Richards	agent	$\varnothing$
		Paint It Black	object	$\varnothing$
21-06-1967	record	Paint It black	object	$\varnothing$
		The Rolling Stones	agent	$\varnothing$
23-06-1967	produce	$\left\langle 21\text{-}06\text{-}1967, \left\langle \text{record}, \left\{ \begin{array}{l} \langle \text{Paint It black}, \text{object}, \varnothing \rangle, \\ \langle \text{The Rolling Stones}, \text{agent}, \varnothing \rangle \end{array} \right\} \right\rangle \right\rangle$	object	$\left\{ \begin{array}{l} \text{tape number 666}, \\ \text{studio number 11} \end{array} \right\}$
		Mick Jagger	agent	$\varnothing$
		Keith Richards	agent	$\varnothing$
12-12-1989	writes	I Want You	object	$\varnothing$
		A. Knijff	agent	$\varnothing$
10-02-1989	set up	P. Frederiks	agent	$\varnothing$
		A. Knijff	agent	$\varnothing$
		The Playful Plebs	object	$\varnothing$
29-04-1991	record	Playful Plebs	agent	$\varnothing$
		I Want You	object	$\varnothing$
29-04-1991	record	Playful Plebs	agent	$\varnothing$
		Long Way To Go	object	$\varnothing$
05-05-1991	produces	H. Honer	agent	$\varnothing$
		$\left\langle 29\text{-}04\text{-}1991, \left\langle \text{record}, \left\{ \begin{array}{l} \langle \text{I Want You}, \text{object}, \varnothing \rangle, \\ \langle \text{Playful Plebs}, \text{agent}, \varnothing \rangle \end{array} \right\} \right\rangle \right\rangle$	object	$\left\{ \begin{array}{l} \text{tape number 3}, \\ \text{studio number 2} \end{array} \right\}$

Table B.1: A sample population of figure 4.1

Appendix C

Graphical Symbols

Sometimes I'm up, sometimes I'm down
Sometimes I'm falling on the ground
Why do you hide, why do you hide your love baby

From: "Hide Your Love",
The Rolling Stones

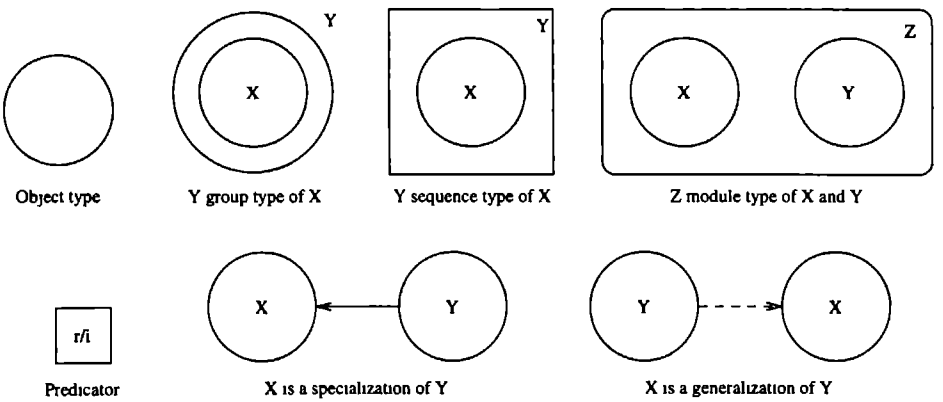


Figure C.1: Symbols of object action involvement model

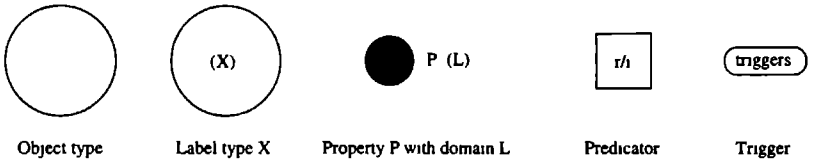


Figure C.2: Symbols of object property model

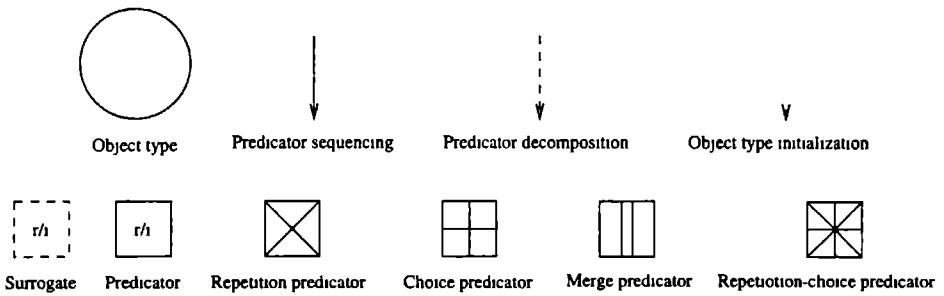


Figure C.3: Symbols of object life model

Appendix D

Sample Information Grammar

Everyday I need another dose
I can't stand it when the music stops
Hot stuff

From "Hot Stuff",
The Rolling Stones

This appendix describes a a complete information grammar of the running example of this thesis. This grammar is obtained by performing all pseudo code procedures described in chapter 5, and is implemented with the AGFL formalism ([Kos91]) as described and demonstrated in chapter 6. Note that a more efficient grammar can be obtained applying all kinds of grammar transformations.

D.1 Domain independent meta rules

```
R :: r ; i.
NUMBER :: singular ; plural.
MODE :: actual ; structural ; trigger ; life.
SORT :: typed ; instantiated.
PREF :: one ; two.
```

D.2 Domain dependent meta rules

D.2.1 Predicators of action types

```
P_I :: band.
P_II :: song.
P_III :: recording.
P_IV :: producer ; band ; person ; musician.
```

```
P_V :: song.
P_VI :: person ; musician.
P_VII :: band.
P_VIII :: person ; musician.
P_IX :: band.
P_X :: person ; musician.
P_XI :: band.
P_XII :: musician.
P_XIII :: musician.
P_XIV :: band.
P_XV :: person.
P_XVI :: person ; musician.
```

D.2.2 Predicators of retrieval and update action types

```
P_XVII :: band.
P_XVIII :: band.
P_XIX :: name.
P_XX :: name.
P_XXI :: person ; musician.
P_XXII :: person ; musician.
P_XXIII :: name.
P_XXIV :: name.
P_XXV :: person ; musician.
P_XXVI :: person ; musician.
P_XXVII :: date.
P_XXVIII :: date.
P_XXIX :: person ; musician.
P_XXX :: person ; musician.
P_XXXI :: name.
P_XXXII :: name.
P_XXXIII :: recording.
P_XXXIV :: recording.
P_XXXV :: number.
P_XXXVI :: number.
P_XXXVII :: recording.
P_XXXVIII :: recording.
P_XXXIX :: number.
P_XXXX :: number.
P_XXXXI :: musician.
P_XXXXII :: musician.
P_XXXXIII :: name.
P_XXXXIV :: name.
```

D.2.3 Module properties

```
ALL_PROPS_MAIN_MODULE :: band name ;
```

```
                    instrument ;
                    person name ; birth date ;
                    tape number ; studio number ;
                    song name.
```

D.3 Driver

```
START : MAIN.

MAIN : MAIN(actual) ; MAIN(structural) ; MAIN(trigger) ; MAIN(life).

MAIN(actual) : TO DISBAND, ".",
               TO JOIN, ".",
               TO LEAVE, ".",
               TO PERFORM, ".",
               TO PRODUCE, ".",
               TO RECORD, ".",
               TO SET UP, ".",
               TO SUBSCRIBE, ".",
               TO UNSUBSCRIBE, ".",
               TO WRITE, ".",
               TO ASK_BAND NAME, ".",
               TO ASK_BIRTH DATE, ".",
               TO ASK_INSTRUMENT NAME, ".",
               TO ASK_PERSON NAME, ".",
               TO ASK_SONG NAME, ".",
               TO ASK_STUDIO NUMBER, ".",
               TO ASK_TAPE NUMBER, ".",
               TO GET_BAND NAME, ".",
               TO GET_BIRTH DATE, ".",
               TO GET_INSTRUMENT NAME, ".",
               TO GET_PERSON NAME, ".",
               TO GET_SONG NAME, ".",
               TO GET_STUDIO NUMBER, ".",
               TO GET_TAPE NUMBER, ".".

MAIN(structural) : GEN_MAIN_MODULE,
                   SPEC_MAIN_MODULE,
                   GROUP_MAIN_MODULE,
                   MOD_MAIN_MODULE,
                   PROPS_MAIN_MODULE.

MAIN(trigger) : TO GET_BAND NAME(band name), ".",
                TO GET_BIRTH DATE(birth date), ".",
                TO GET_INSTRUMENT NAME(instrument), ".",
                TO GET_PERSON NAME(person name), ".",
```

```
                    TO GET_SONG NAME(song name), ".",
                    TO GET_STUDIO NUMBER(studio number), ".",
                    TO GET_TAPE NUMBER(tape number), ".".

MAIN(life) : LIFE OF BAND,
             LIFE OF MUSICIAN,
             LIFE OF PERSON,
             LIFE OF PRODUCER,
             LIFE OF RECORDING,
             LIFE OF SONG.
```

D.4 Object types

D.4.1 Abstract object types

```
BAND : BAND(one) ; BAND(two).
BAND(one) : "Band".
BAND(two) : "Pop group".

MAIN_MODULE : "Main".

MUSICIAN(singular) : MUSICIAN.
MUSICIAN : "Musician".
MUSICIAN(plural): "Musicians".

PERSON : "Person".
PRODUCER : "Producer".

RECORDING : RECORDING(one) ; RECORDING(two).
RECORDING(one) : "Recording".
RECORDING(two) : "the recording of", P_II(song), "recorded by", P_I(band).

SONG : "Song".
```

D.4.2 Concrete object types

```
DATE : "Date".

NAME : "String".

NUMBER : "Number".
```

D.5 Action types

D.5.1 Action types concerning abstract object types

```
TO DISBAND : TO DISBAND(r).
TO DISBAND(r) : TO DISBAND(r,band).
TO DISBAND(r,band) : BAND, "disbands".

TO JOIN : TO JOIN(r) ; TO JOIN(i).
TO JOIN(r) : TO JOIN(r,band,person) ; TO JOIN(r,band,musician).
TO JOIN(i) : TO JOIN(i,band,person) ; TO JOIN(i,band,musician).

TO JOIN(r,band,person) : P_X(person), "joins", P_IX(band).
TO JOIN(r,band,musician) : P_X(musician), "joins", P_IX(band).

TO JOIN(i,band,person) : P_IX(band), "is joined by", P_X(person).
TO JOIN(i,band,musician) : P_IX(band), "is joined by", P_X(musician).

TO LEAVE : TO LEAVE(r) ; TO LEAVE(i).
TO LEAVE(r) : TO LEAVE(r,band,musician).
TO LEAVE(i) : TO LEAVE(i,band,musician).

TO LEAVE(r,band,musician) : P_X(musician), "leaves", P_IX(band).
TO LEAVE(i,band,musician) : P_IX(band), "is left by", P_X(musician).

TO PERFORM : TO PERFORM(r).
TO PERFORM(r) : TO PERFORM(r,musician).

TO PERFORM(r,musician) : P_XIII(musician), "performs".

TO PRODUCE : TO PRODUCE(r) ; TO PRODUCE(i).
TO PRODUCE(r) : TO PRODUCE(r,recording,producer) ;
                TO PRODUCE(r,recording,band) ;
                TO PRODUCE(r,recording,person) ;
                TO PRODUCE(r,recording,musician).
TO PRODUCE(i) : TO PRODUCE(i,recording,producer) ;
                TO PRODUCE(i,recording,band) ;
                TO PRODUCE(i,recording,person) ;
                TO PRODUCE(i,recording,musician).

TO PRODUCE(r,recording,producer) :
  P_IV(producer), "produces", P_III(recording).
TO PRODUCE(r,recording,band) :
  P_IV(band), "produces", P_III(recording).
TO PRODUCE(r,recording,person) :
  P_IV(person), "produces", P_III(recording).
```

```
TO PRODUCE(r,recording,musician) :
  P_IV(musician), "produces", P_III(recording).

TO PRODUCE(i,recording,producer) :
  P_III(recording), "is produced by", P_IV(producer).
TO PRODUCE(i,recording,band) :
  P_III(recording), "is produced by", P_IV(band).
TO PRODUCE(i,recording,person) :
  P_III(recording), "is produced by", P_IV(person).
TO PRODUCE(i,recording,musician) :
  P_III(recording), "is produced by", P_IV(musician).

TO RECORD : TO RECORD(r) ; TO RECORD(i).
TO RECORD(r) : TO RECORD(r,song,band).
TO RECORD(i) : TO RECORD(i,song,band).

TO RECORD(r,song,band) : P_I(band), "records", P_II(song).

TO RECORD(i,song,band) : P_II(song), "is recorded by", P_I(band).

TO SET UP : TO SET UP(r) ; TO SET UP(i).
TO SET UP(r) : TO SET UP(r,band,person) ; TO SET UP(r,band,musician).
TO SET UP(i) : TO SET UP(i,band,person) ; TO SET UP(i,band,musician).

TO SET UP(r,band,person) :
  P_VIII(person), "sets up", P_VII(band).
TO SET UP(r,band,musician) :
  P_VIII(musician), "sets up", P_VII(band).

TO SET UP(i,band,person) :
  P_VII(band), "is set up by", P_VIII(person).
TO SET UP(i,band,musician) :
  P_VII(band), "is set up by", P_VIII(musician).

TO SUBSCRIBE : TO SUBSCRIBE(i).
TO SUBSCRIBE(i) : TO SUBSCRIBE(i,person).

TO SUBSCRIBE(i,person) : P_XV(person), "subscribes".

TO UNSUBSCRIBE : TO UNSUBSCRIBE(r).
TO UNSUBSCRIBE(r) : TO UNSUBSCRIBE(r,person) ; TO UNSUBSCRIBE(r,musician).

TO UNSUBSCRIBE(r,person) : P_XVI(person), "unsubscribes".
TO UNSUBSCRIBE(r,musician) : P_XVI(musician), "unsubscribes".

TO WRITE : TO WRITE(r) ; TO WRITE(i).
```

```
TO WRITE(r) : TO WRITE(r,song,person) ; TO WRITE(r,song,musician).
TO WRITE(i) : TO WRITE(i,song,person) ; TO WRITE(i,song,musician).

TO WRITE(r,song,person) :
  P_VI(person), "writes", P_V(song).
TO WRITE(r,song,musician) :
  P_VI(musician), "writes", P_V(song).

TO WRITE(i,song,person) :
  P_V(song), "is written by", P_VI(person).
TO WRITE(i,song,musician) :
  P_V(song), "is written by", P_VI(musician).

P_I(band) : BAND.
P_II(song) : SONG.
P_III(recording) : RECORDING.
P_IV(producer) : PRODUCER.
P_IV(band) : BAND.
P_IV(person) : PERSON.
P_IV(musician) : MUSICIAN.
P_V(song) : SONG.
P_VI(person) : PERSON.
P_VI(musician) : MUSICIAN.
P_VII(band) : BAND.
P_VIII(person) : PERSON.
P_VIII(musician) : MUSICIAN.
P_IX(band) : BAND.
P_X(person) : PERSON.
P_X(musician) : MUSICIAN.
P_XI(band) : BAND.
P_XII(musician) : MUSICIAN.
P_XIII(musician) : MUSICIAN.
P_XIV(band) : BAND.
P_XV(person) : PERSON.
P_XVI(person) : PERSON.
P_XVI(musician) : MUSICIAN.
```

D.5.2 Retrieval and update action types

```
TO ASK_BAND NAME : TO ASK_BAND NAME(r) ; TO ASK_BAND NAME(i).
TO ASK_BAND NAME(r) : TO ASK_BAND NAME(r,band,name).
TO ASK_BAND NAME(i) : TO ASK_BAND NAME(i,band,name).

TO ASK_BAND NAME(r,band,name) : P_XVIII(band), ASKS, BAND NAME.

TO ASK_BAND NAME(i,band,name) : BAND NAME, BELONGS TO, P_XVII(band).
```

```
TO ASK_BIRTH DATE : TO ASK_BIRTH DATE(r) ; TO ASK_BIRTH DATE(ı).
TO ASK_BIRTH DATE(r) : TO ASK_BIRTH DATE(r,person,date) ;
                       TO ASK_BIRTH DATE(r,musician,date).
TO ASK_BIRTH DATE(ı) : TO ASK_BIRTH DATE(ı,person,date) ;
                       TO ASK_BIRTH DATE(ı,musician,date).

TO ASK_BIRTH DATE(r,person,date) :
  P_XXV(person), ASKS, BIRTH DATE.
TO ASK_BIRTH DATE(r,musician,date) :
  P_XXV(musician), ASKS, BIRTH DATE.

TO ASK_BIRTH DATE(ı,person,date) :
  BIRTH DATE, BELONGS TO, P_XXV(person).
TO ASK_BIRTH DATE(ı,musician,date) :
  BIRTH DATE, BELONGS TO, P_XXV(musician).

TO ASK_INSTRUMENT NAME :
  TO ASK_INSTRUMENT NAME(r) ; TO ASK_INSTRUMENT NAME(ı).
TO ASK_INSTRUMENT NAME(r) :
  TO ASK_INSTRUMENT NAME(r,musician,name).
TO ASK_INSTRUMENT NAME(ı) :
  TO ASK_INSTRUMENT NAME(ı,musician,name).

TO ASK_INSTRUMENT NAME(r,musician,name) :
  P_XXXXI(musician), ASKS, INSTRUMENT NAME.

TO ASK_INSTRUMENT NAME(ı,musician,name) :
  INSTRUMENT NAME, BELONGS TO, P_XXXXI(musician).

TO ASK_PERSON NAME : TO ASK_PERSON NAME(r) ; TO ASK_PERSON NAME(ı).
TO ASK_PERSON NAME(r) : TO ASK_PERSON NAME(r,person,name) ;
                        TO ASK_PERSON NAME(r,musician,name).
TO ASK_PERSON NAME(ı) : TO ASK_PERSON NAME(ı,person,name) ;
                        TO ASK_PERSON NAME(ı,musician,name).

TO ASK_PERSON NAME(r,person,name) :
  P_XXI(person), ASKS, PERSON NAME.
TO ASK_PERSON NAME(r,musician,name) :
  P_XXI(musician), ASKS, PERSON NAME.

TO ASK_PERSON NAME(ı,person,name) :
  PERSON NAME, BELONGS TO, P_XXI(person).
TO ASK_PERSON NAME(ı,musician,name) :
  PERSON NAME, BELONGS TO, P_XXI(musician).
```

```
TO ASK_SONG NAME : TO ASK_SONG NAME(r) ; TO ASK_SONG NAME(i).
TO ASK_SONG NAME(r) : TO ASK_SONG NAME(r,song,name).
TO ASK_SONG NAME(i) : TO ASK_SONG NAME(i,song,name).

TO ASK_SONG NAME(r,song,name) : P_XXIX(song), ASKS, SONG NAME.

TO ASK_SONG NAME(i,song,name) :
  SONG NAME, BELONGS TO, P_XXIX(song).

TO ASK_STUDIO NUMBER :
  TO ASK_STUDIO NUMBER(r) ; TO ASK_STUDIO NUMBER(i).
TO ASK_STUDIO NUMBER(r) :
  TO ASK_STUDIO NUMBER(r,recording,number).
TO ASK_STUDIO NUMBER(i) .
  TO ASK_STUDIO NUMBER(i,recording,number).

TO ASK_STUDIO NUMBER(r,recording,number) :
  P_XXXVII(recording), ASKS, STUDIO NUMBER.

TO ASK_STUDIO NUMBER(i,recording,number) :
  STUDIO NUMBER, BELONGS TO, P_XXXVII(recording).

TO ASK_TAPE NUMBER :
  TO ASK_TAPE NUMBER(r) ; TO ASK_TAPE NUMBER(i).
TO ASK_TAPE NUMBER(r) :
  TO ASK_TAPE NUMBER(r,recording,number).
TO ASK_TAPE NUMBER(i) :
  TO ASK_TAPE NUMBER(i,recording,number).

TO ASK_TAPE NUMBER(r,recording,number) :
  P_XXXIII(recording), ASKS, TAPE NUMBER.

TO ASK_TAPE NUMBER(i,recording,number) :
  TAPE NUMBER, BELONGS TO, P_XXXIII(recording).

TO GET_BAND NAME : TO GET_BAND NAME(r) ; TO GET_BAND NAME(i).
TO GET_BAND NAME(r) : TO GET_BAND NAME(r,band,name).
TO GET_BAND NAME(i) : TO GET_BAND NAME(i,band,name).

TO GET_BAND NAME(r,band,name) : P_XVIII(band), GETS, BAND NAME.

TO GET_BAND NAME(i,band,name) : BAND NAME, IS GIVEN TO, P_XVIII(band).

TO GET_BIRTH DATE : TO GET_BIRTH DATE(r) ; TO GET_BIRTH DATE(i).
TO GET_BIRTH DATE(r) : TO GET_BIRTH DATE(r,person,date) ;
                       TO GET_BIRTH DATE(r,musician,date).
```

```
TO GET_BIRTH DATE(i) : TO GET_BIRTH DATE(i,person,date) ;
                       TO GET_BIRTH DATE(i,musician,date).

TO GET_BIRTH DATE(r,person,date) :
  P_XXVI(person), GETS, BIRTH DATE.
TO GET_BIRTH DATE(r,musician,date) :
  P_XXVI(musician), GETS, BIRTH DATE.

TO GET_BIRTH DATE(i,person,date) :
  BIRTH DATE, IS GIVEN TO, P_XXVI(person).
TO GET_BIRTH DATE(i,musician,date) :
  BIRTH DATE, IS GIVEN TO, P_XXVI(musician).

TO GET_INSTRUMENT NAME :
  TO GET_INSTRUMENT NAME(r) ; TO GET_INSTRUMENT NAME(i).
TO GET_INSTRUMENT NAME(r) :
  TO GET_INSTRUMENT NAME(r,musician,name).
TO GET_INSTRUMENT NAME(i) :
  TO GET_INSTRUMENT NAME(i,musician,name).

TO GET_INSTRUMENT NAME(r,musician,name) :
  P_XXXXII(musician), GETS, INSTRUMENT NAME.

TO GET_INSTRUMENT NAME(i,musician,name) :
  INSTRUMENT NAME, IS GIVEN TO, P_XXXXII(musician).

TO GET_PERSON NAME : TO GET_PERSON NAME(r) ; TO GET_PERSON NAME(i).
TO GET_PERSON NAME(r) : TO GET_PERSON NAME(r,person,name) ;
                        TO GET_PERSON NAME(r,musician,name).
TO GET_PERSON NAME(i) : TO GET_PERSON NAME(i,person,name) ;
                        TO GET_PERSON NAME(i,musician,name).

TO GET_PERSON NAME(r,person,name) :
  P_XXII(person), GETS, PERSON NAME.
TO GET_PERSON NAME(r,musician,name) :
  P_XXII(musician), GETS, PERSON NAME.

TO GET_PERSON NAME(i,person,name) :
  PERSON NAME, IS GIVEN TO, P_XXII(person).
TO GET_PERSON NAME(i,musician,name) :
  PERSON NAME, IS GIVEN TO, P_XXII(musician).

TO GET_SONG NAME : TO GET_SONG NAME(r) ; TO GET_SONG NAME(i).
TO GET_SONG NAME(r) : TO GET_SONG NAME(r,song,name).
TO GET_SONG NAME(i) : TO GET_SONG NAME(i,song,name).
```

```
TO GET_SONG NAME(r,song,name) : P_XXX(song), GETS, SONG NAME.

TO GET_SONG NAME(i,song,name) :
  SONG NAME, IS GIVEN TO, P_XXX(song).

TO GET_STUDIO NUMBER :
  TO GET_STUDIO NUMBER(r) ; TO GET_STUDIO NUMBER(i).
TO GET_STUDIO NUMBER(r) :
  TO GET_STUDIO NUMBER(r,recording,number).
TO GET_STUDIO NUMBER(i) :
  TO GET_STUDIO NUMBER(i,recording,number).

TO GET_STUDIO NUMBER(r,recording,number) :
  P_XXXVIII(recording), GETS, STUDIO NUMBER.

TO GET_STUDIO NUMBER(i,recording,number) :
  STUDIO NUMBER, IS GIVEN TO, P_XXXVIII(recording).

TO GET_TAPE NUMBER : TO GET_TAPE NUMBER(r) ; TO GET_TAPE NUMBER(i).
TO GET_TAPE NUMBER(r) : TO GET_TAPE NUMBER(r,recording,number).
TO GET_TAPE NUMBER(i) : TO GET_TAPE NUMBER(i,recording,number).

TO GET_TAPE NUMBER(r,recording,number) :
  P_XXXIV(recording), GETS, TAPE NUMBER.

TO GET_TAPE NUMBER(i,recording,number) :
  TAPE NUMBER, IS GIVEN TO, P_XXXIV(recording).

P_XVII(band) : BAND.
P_XVIII(band) : BAND.
P_XIX(name) : NAME.
P_XX(name) : NAME.
P_XXI(person) : PERSON.
P_XXI(musician) : MUSICIAN.
P_XXII(person) : PERSON.
P_XXII(musician) : MUSICIAN.
P_XXIII(name) : NAME.
P_XXIV(name) : NAME.
P_XXV(person) : PERSON.
P_XXV(musician) : MUSICIAN.
P_XXVI(person) : PERSON.
P_XXVI(musician) : MUSICIAN.
P_XXVII(date) : DATE.
P_XXVIII(date) : DATE.
P_XXIX(song) : SONG.
P_XXX(song) : SONG.
```

```
P_XXXI(name) : NAME.
P_XXXII(name) : NAME.
P_XXXIII(recording) : RECORDING.
P_XXXIV(recording) : RECORDING.
P_XXXV(number) : NUMBER.
P_XXXVI(number) : NUMBER.
P_XXXVII(recording) : RECORDING.
P_XXXVIII(recording) : RECORDING.
P_XXXIX(number) : NUMBER.
P_XXXX(number) : NUMBER.
P_XXXXI(musician) : MUSICIAN.
P_XXXXII(musician) : MUSICIAN.
P_XXXXIII(name) : NAME.
P_XXXXIV(name) : NAME.
```

D.6 Properties

```
BAND NAME : "Band name".
BIRTH DATE : "Birth date".
INSTRUMENT NAME : "Instrument".
PERSON NAME : "Person name".
SONG NAME : "Song name".
STUDIO NUMBER : "Studio number".
TAPE NUMBER : "Tape number".
```

D.7 Triggers

```
TO GET_BAND NAME(band name) : "when", TO SET UP,
                              "then", TO GET_BAND NAME.

TO GET_BIRTH DATE(birth date) : "when", TO SUBSCRIBE,
                                "then", TO GET_BIRTH DATE.

TO GET_INSTRUMENT NAME(instrument) : "when", TO SET UP,
                                     "then", TO GET_INSTRUMENT NAME.

TO GET_INSTRUMENT NAME(instrument) : "when", TO JOIN,
                                     "then", TO GET_INSTRUMENT NAME.

TO GET_PERSON NAME(person name) : "when", TO SUBSCRIBE,
                                  "then", TO GET_PERSON NAME.

TO GET_SONG NAME(song name) : "when", TO WRITE,
                              "then", TO GET_SONG NAME.
```

```
TO GET_STUDIO NUMBER(studio number) : "when", TO RECORD,
                                      "then", TO GET_STUDIO NUMBER.

TO GET_TAPE NUMBER(tape number) : "when", TO RECORD,
                                  "then", TO GET_TAPE NUMBER.
```

D.8 Structure

```
GEN_MAIN_MODULE : GEN_PRODUCER_BAND, GEN_PRODUCER_PERSON.
GEN_PRODUCER_BAND : BAND, "is a", PRODUCER, ".".
GEN_PRODUCER_PERSON : PERSON, "is a", PRODUCER, ".".

SPEC_MAIN_MODULE : SPEC_MUSICIAN_PERSON.
SPEC_MUSICIAN_PERSON : MUSICIAN, "is a", PERSON, ".".

GROUP_MAIN_MODULE : GROUP_BAND.
GROUP_BAND : BAND, "is a group of", MUSICIAN(plural), ".".

MOD_MAIN_MODULE : MAIN_MODULE, "is a composition of",
                  "Band", ",", "Musician", ",", "Person", ",",
                  "Producer", ",", "Recording", ",", "Song", ",",
                  "to disband", ",", "to join", ",", "to leave", ",",
                  "to perform", ",", "to produce", ",", "to record", ",",
                  "to set up", ",", "to subscribe", ",",
                  "to unsubscribe", ",", "to write", ",",
                  "to ask Band name", ",", "to ask Birth date", ",",
                  "to ask Instrument name", ",", "to ask Person name", ",",
                  "to ask Song name", ",", "to ask Studio number", ",",
                  "to ask Tape number", ",",
                  "to get Band name", ",", "to get Birth date", ",",
                  "to get Instrument name", ",", "to get Person name", ",",
                  "to get Song name", ",", "to get Studio number", ",",
                  "to get Tape number", ",",
                  "String", ",", "Date", ",", "Number", ".".

PROPS_MAIN_MODULE : PROPS_BAND,
                    PROPS_MUSICIAN,
                    PROPS_PERSON,
                    PROPS_RECORDING,
                    PROPS_SONG.

PROPS_BAND : BAND, HAS, BAND NAME, DENOTED AS, NAME, ".".

PROPS_MUSICIAN : MUSICIAN, HAS, PERSON NAME, DENOTED AS, NAME, ".",
                 MUSICIAN, HAS, BIRTH DATE, DENOTED AS, DATE, ".",
                 MUSICIAN, HAS, INSTRUMENT NAME, DENOTED AS, NAME, ".".
```

```
PROPS_PERSON : PERSON, HAS, PERSON NAME, DENOTED AS, NAME, ".",
               PERSON, HAS, BIRTH DATE, DENOTED AS, DATE, ".".

PROPS_RECORDING : RECORDING, HAS, STUDIO NUMBER, DENOTED AS, NUMBER, ".",
                  RECORDING, HAS, TAPE NUMBER, DENOTED AS, NUMBER, ".".

PROPS_SONG : SONG, HAS, SONG NAME, DENOTED AS, NAME, ".".
```

D.9 Course of life

```
LIFE OF BAND : SEQ_C_I.

SEQ_C_I : TO SET UP(i), ".",
          THEN, SEQ_D_I.

SEQ_D_I : REPEATEDLY, FRAG_D_I, OR, FRAG_D_II, OR,
                      FRAG_D_III, OR, FRAG_D_IV, ".",
          THEN, SEQ_D_II.

SEQ_D_II : TO DISBAND, ".".

FRAG_D_I   : TO RECORD(r,song,band).
FRAG_D_II  : TO LEAVE(i).
FRAG_D_III : TO JOIN(i).
FRAG_D_IV  : TO PRODUCE(r,recording,band).

LIFE OF MUSICIAN : SEQ_C_II.

SEQ_C_II : FRAG_D_V, OR, FRAG_D_VI, ".",
           THEN, SEQ_D_III.

SEQ_D_III : REPEATEDLY, FRAG_D_VII, OR, FRAG_D_VIII, ".".

FRAG_D_V    : TO SET UP(r,band,person).
FRAG_D_VI   : TO JOIN(r,band,person).
FRAG_D_VII  : TO PERFORM.
FRAG_D_VIII : TO LEAVE(r).

LIFE OF PERSON : SEQ_C_III.

SEQ_C_III : TO SUBSCRIBE, ".",
            THEN, SEQ_D_IV.

SEQ_D_IV : REPEATEDLY, FRAG_D_IX, OR, FRAG_D_X, OR,
                       FRAG_D_XI, OR, FRAG_D_XII, ".",
```

```
            THEN, SEQ_D_V.

SEQ_D_V : TO UNSUBSCRIBE(r,person), ".".

FRAG_D_IX  : TO WRITE(r,song,person).
FRAG_D_X   : TO SET UP(r,band,person).
FRAG_D_XI  : TO JOIN(r,band,person).
FRAG_D_XII : TO PRODUCE(r,recording,person).

LIFE OF PRODUCER : SEQ_C_IV.

SEQ_C_IV: FRAG_D_XIII, OR, FRAG_D_XIV, ".",
          THEN, SEQ_D_VI.

SEQ_D_VI : REPEATEDLY, FRAG_D_XV, ".",
           THEN, SEQ_D_VII.

SEQ_D_VII : FRAG_D_XVI, OR, FRAG_D_XVII, ".".

FRAG_D_XIII : TO SUBSCRIBE.
FRAG_D_XIV  : TO SET UP.
FRAG_D_XV   : TO PRODUCE.
FRAG_D_XVI  : TO UNSUBSCRIBE.
FRAG_D_XVII : TO DISBAND.

LIFE OF RECORDING : SEQ_C_V.

SEQ_C_V : TO RECORD, ".",
          THEN, SEQ_D_VIII.

SEQ_D_VIII : REPEATEDLY, FRAG_D_XVIII, ".".

FRAG_D_XVIII : TO PRODUCE(i).

LIFE OF SONG : SEQ_C_VI.

SEQ_C_VI : TO WRITE(i), ".",
           THEN, SEQ_D_IX.

SEQ_D_IX : REPEATEDLY, FRAG_D_XIX, ".".

FRAG_D_XIX : TO RECORD(i).
```

D.10 Auxiliary rules

```
ASKS : "asks".
```

```
BELONGS TO : "belongs to".

GETS : "gets".
IS GIVEN TO : "is given to".

HAS : "has".
DENOTED AS : "denoted as".

THEN : "Then".
REPEATEDLY : "repeatedly".
OR : "or".
```

Appendix E

Process Algebra

Did you ever wake up to find
A day that broke up your mind
Destroyed your notions of circular time

From: "Sway",
The Rolling Stones

The name *process algebra* ([BW90a]) actually refers to a whole family of algebras based on a same principle. Traditionally, only the family member used is presented.

The basic units of process algebra are the *atomic actions*. From the atomic actions one can build new processes with *alternative composition* (*choice*) and *sequential composition* ($\oplus$ respectively. $\odot$). The algebra that results is called basic process algebra (BPA). Table E.1 summarizes the axioms of BPA.

$$
\begin{aligned}
X \oplus Y &= Y \oplus X && \text{(BPA1)}\\
(X \oplus Y) \oplus Z &= X \oplus (Y \oplus Z) && \text{(BPA2)}\\
X \oplus X &= X && \text{(BPA3)}\\
(X \oplus Y) \odot Z &= X \odot Z \oplus Y \odot Z && \text{(BPA4)}\\
(X \odot Y) \odot Z &= X \odot (Y \odot Z) && \text{(BPA5)}
\end{aligned}
$$

Table E.1: BPA

We consider a special constant the *empty action*, denoted as ε. The empty action is used to denote a process that does nothing but terminate. Another special constant δ, *deadlock*, denotes an action which does not terminate (see also table E.2).

$$
\begin{aligned}
X \odot \varepsilon &= X && \text{(BPA6)}\\
\varepsilon \odot X &= X && \text{(BPA7)}\\
X \oplus \delta &= X && \text{(BPA8)}\\
\delta \odot X &= \delta && \text{(BPA9)}
\end{aligned}
$$

Table E.2: δ en ε axioms

$$
\begin{aligned}
X \parallel Y &= X \between Y \oplus Y \between X \oplus \surd(X)\surd(Y) && \text{(TM1)}\\
\varepsilon \between X &= \delta && \text{(TM2)}\\
aX \between Y &= a(X \parallel Y) && \text{(TM3)}\\
(X \oplus Y) \between Z &= X \between Z \oplus Y \between Z && \text{(TM4)}
\end{aligned}
$$

Table E.3: Axioms of merge operator

Another constructor is the so-called *merge operator*, $\parallel$, which is used to describe parallelism (see table E.3). The merge operator is defined with aid of an auxiliary operator, the *left-merge operator* ($\between$). Finally the *termination operator* $\surd$ determines whether or not a direct termination option (ε) is present for a given process (see table E.4).

$$
\begin{aligned}
\surd(\varepsilon) &= \varepsilon && \text{(TE1)}\\
\surd(a) &= \delta && \text{(TE2)}\\
\surd(X \oplus Y) &= \surd(X) \oplus \surd(Y) && \text{(TE3)}\\
\surd(X \odot Y) &= \surd(X) \odot \surd(Y) && \text{(TE4)}
\end{aligned}
$$

Table E.4: Termination operator

Table E.5 shows the axioms for obtaining a set of traces for process algebra expressions.

As a convention the names of the atomic actions are written in lowercase while process variables are written in uppercase.

$$\begin{aligned}
\mathsf{Traces}(\varepsilon) &= \{\lambda\} && \text{(TR1)}\\
\mathsf{Traces}(a) &= \{\lambda, a\} && \text{(TR2)}\\
\mathsf{Traces}(a \odot X) &= \{\lambda\} \cup \{a \cdot \sigma \mid \sigma \in \mathsf{Traces}(X)\} && \text{(TR3)}\\
\mathsf{Traces}(X \oplus Y) &= \mathsf{Traces}(X) \oplus \mathsf{Traces}(Y) && \text{(TR4)}
\end{aligned}$$

Table E.5: Trace axioms

Appendix F

Category Theory

I think I've had enough
You know religion is though
It's a state of mind I don't need

From: "Send It To Me",
The Rolling Stones

This appendix[1] contains the definitions of the categorical constructs and notations needed in this thesis, in order to make it self-contained as much as possible. For an in-depth treatment of category theory the reader is referred to [BW90b].

F.1 Basics

A directed multigraph is a directed graph where there may be multiple edges with the same direction between two nodes.

Definition F.1

A *directed multigraph* G consists of a set of nodes G_0 and a set of edges G_1. The source and target of an edge can be found by application of the functions source and target respectively. The notation $f : A \to B$ implies that f is an edge with $\mathsf{source}(f) = A$ and $\mathsf{target}(f) = B$.

The following definition defines a category as a special kind of multigraph.

Definition F.2

A *category* C is a directed multigraph whose nodes are called *objects* and whose edges are called *arrows*. For each pair of arrows $f : A \to B$ and $g : B \to C$ there is an associated arrow $g \circ f : A \to C$, the *composition* of f with g. Furthermore,

[1] Most of the definitions in this appendix are adapted from [BW90b].

$(h \circ g) \circ f = h \circ (g \circ f)$ whenever either side is defined. For each object A there is an arrow $\mathsf{Id}_A : A \to A$, the *identity* arrow. If $f : A \to B$, then $f \circ \mathsf{Id}_A = f = \mathsf{Id}_B \circ f$.

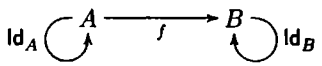


Figure F.1: A simple example of a category

In figure F.1 a simple example of a category is shown. It is an abstract example, no assumptions about the meaning of the objects and the arrows have been made (and indeed, have to be made!). In this category the choice of composites is forced: $f \circ \mathsf{Id}_A = f = \mathsf{Id}_B \circ f$. In category theory it is customary to omit the identity arrows in drawings of categories if they do not serve a particular purpose. We will adopt this convention in this thesis. The objects and arrows of a category may also have a concrete interpretation. For example, objects may be mathematical structures such as sets, partially ordered sets, graphs, trees etc. Arrows can denote functions, relations, paths in a graph, etc.

As a concrete example of a category in the context of information systems consider the set of all instantiations of a data base, and all possible updates on these instantiations. The instantiations may serve as objects, and the updates as arrows of the corresponding category. Each object has an identity arrow, if one considers the "neutral" update, i.e. the update that does not change an instantiation at all, to be a normal update. One can easily verify that this indeed constitutes a category. Arrow composition is associative as update composition is associative. Also, the neutral update serves as a neutral element with respect to arrow composition: an update composed with a neutral update simply yields that update.

In the context of this thesis, some set-oriented categories are important. The most elementary and frequently used category is the category **Set**, where the objects are sets and the arrows are total functions. The objects of **Set** are not necessarily finite. The category whose objects are *finite* sets and whose arrows are total functions is called **FinSet**. The category **PartSet** concerns sets with *partial* functions, while the category **Rel** has sets as objects and binary *relations* as arrows.

Some arrows have special properties. We consider three important kinds of arrows: *monomorphisms*, *epimorphisms* and *isomorphisms*.

Definition F.3

An arrow $f : A \to B$ is a *monomorphism* if for any object X of the category and any arrows $x, y : X \to A$, if $f \circ x = f \circ y$, then $x = y$.

Figure F.2 illustrates the definition of a monomorphism. A monomorphism in the category **Set** captures the idea of an injective function. In the category **PartSet** a monomorphism describes a total and injective function.

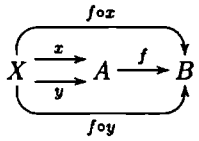


Figure F.2: Illustration of the definition of a monomorphism

Definition F.4

An arrow $f : B \to A$ is an *epimorphism* if for any object X of the category and any arrows $x, y : A \to X$, if $x \circ f = y \circ f$, then $x = y$.

Figure F.3 illustrates the definition of an epimorphism. In the category **Set** an epimorphism corresponds to a surjective function.

An epimorphism is a monomorphism in the *dual category*. A dual category of a category C, denoted as C^{op}, has the same objects as C and as arrows all arrows of C inverted, i.e. if $f : A \to B$ is an arrow in C then $f^{\mathsf{op}} : B \to A$ is an arrow of C^{op}. As a result the composition of arrows in the dual category is defined on the inverted arrows. The concept of duality in category theory is very important as it reduces proof obligations: the dual of a theorem is also a theorem.

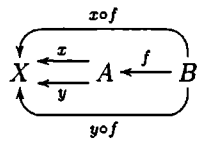


Figure F.3: Illustration of the definition of an epimorphism

The category theoretic equivalent of the set theoretic concept of a bijective function is called an *isomorphism*. In a mathematical context isomorphism means indistinguishable in form. As remarked in [RB88]:

> *Isomorphisms are important in category theory since arrow-theoretic descriptions usually determine an object to within an isomorphism. Thus isomorphisms are the degree of "sameness" that we wish to consider in categories.*

Definition F.5

An arrow $f : A \to B$ is said to be an *isomorphism* if an arrow $g : B \to A$ exists such that $f \circ g = \mathsf{Id}_B$ and $g \circ f = \mathsf{Id}_A$. Arrow f is called the *inverse* of arrow g and vice versa. If such a pair of arrows exists between two objects A and B, A is *isomorphic* with B, which is denoted as $A \cong B$. The identity arrows are the *trivial isomorphisms*.

There are also some objects with special properties.

Definition F.6

An object T of a category C is called a *terminal object* if there is exactly one arrow $A \to T$ for each object A of C. Terminal objects are denoted by 1. The dual notion, an object of a category that has a unique arrow to each object (including itself), is called an *initial object* and denoted as 0.

As terminal (initial) objects are isomorphic, one usually speaks of *the* terminal (initial) object of a certain category.

The initial object in **Set** is the empty set. The terminal objects in **Set** are all singleton sets. In the category **Rel** the empty set is both initial and terminal.

F.2 Diagrams

Many categorical definitions and proofs employ diagrams. As remarked before, quite complex facts can be visualized by the use of these diagrams. The following definition defines what a diagram is.

Definition F.7

Let I and G be graphs. A *diagram* in G of *shape* I is a homomorphism $D : \mathsf{I} \to \mathsf{G}$ of graphs. I is called the *shape graph* of the diagram D.

The following example, taken from [BW90b], illustrates some subtleties involving the concept of diagram.

Example F.1

Let G be a graph with objects A, B, and C and arrows $f : A \to B$, $g : B \to C$, and $h : B \to B$. The following diagram

$$A \xrightarrow{f} B \xrightarrow{g} C$$

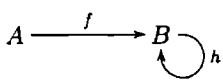
can then be formally defined, using the shape graph I,

$$1 \xrightarrow{u} 2 \xrightarrow{v} 3$$

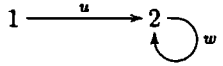
as the homomorphism $D : \mathsf{I} \to \mathsf{G}$ with $D(1) = A$, $D(2) = B$, $D(3) = C$, $D(u) = f$, and $D(v) = g$. The following diagram is just like D (has the same shape) except that v goes to h and 3 goes to B.

$$A \xrightarrow{f} B \xrightarrow{h} B$$

The following diagram has a different shape graph as the two diagrams considered before.



Formally it corresponds to a diagram $E : \mathsf{J} \to \mathsf{G}$, where the shape graph J is defined by



with $E(1) = A$, $E(2) = B$, $E(u) = f$, and $E(w) = h$.

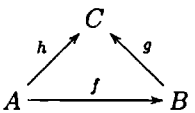
The notion of a *commutative* diagram plays a central role in category theory. Categorical proofs and definitions often use diagrams and prove or require them to commute. Commutative diagrams are the categorist's way of expressing equations.

Definition F.8

A diagram is said to commute if every path between two objects in its image determines through composition the same arrow.

Example F.2

The following diagram commutes if and only if h is the composite $g \circ f$.



F.3 Products and coproducts

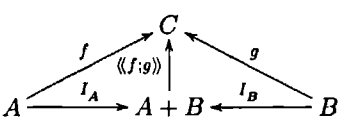
In the disjoint union of a number of sets, elements originating from different sets can always be distinguished. The disjoint union of two sets can be defined in several ways. A possible definition of the disjoint union $A + B$ of two sets A and B is

$$A + B = \big\{\langle a, 0\rangle \;\big|\; a \in A\big\} \cup \big\{\langle b, 1\rangle \;\big|\; b \in B\big\},$$

with canonical injections I_A and I_B, i.e. $I_A(a) = \langle a, 0\rangle$ and $I_B(b) = \langle b, 1\rangle$. The categorical definition of a *coproduct* (also referred to as *sum*) generalizes this definition. In particular, it does not prescribe a representation.

Definition F.9

A *coproduct* of two objects A and B in a category consists of an object $A+B$ together with arrows $I_A : A \to A+B$ and $I_B : B \to A+B$ such that for any arrows $f : A \to C$ and $g : B \to C$, there is a unique arrow, denoted as $\langle\!\langle f; g\rangle\!\rangle : A + B \to C$, for which the following diagram commutes:

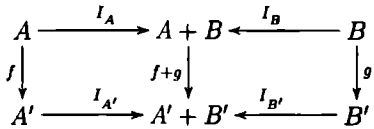


$I_A : A \to A + B$ and $I_B : B \to A + B$ are called *injection arrows* of the sum.

The definition of a coproduct can straightforwardly be generalized to be applicable to any number of objects in a category. Coproducts can also be defined for arrows. In the category **Set**, the coproduct of two arrows $f : A \to A'$ and $g : B \to B'$ is a function $f + g : A + B \to A' + B'$. If this function is applied to an element x of the disjoint union $A+B$ it either yields $f(x)$ or $g(x)$, depending whether x originates from A or B respectively.

Definition F.10

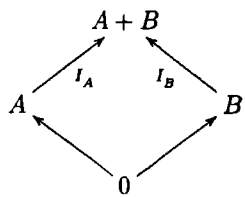
A *coproduct* of two arrows $f : A \to A'$ and $g : B \to B'$ is an arrow $f + g : A + B \to A' + B'$ such that the following diagram commutes:



Sums in the category of sets have special properties they do not have in most other categories. One such property is that sums in **Set** are *disjoint*. In a disjoint sum the sum injection arrows must be monomorphisms.

Definition F.11

Let A and B be two objects in a category with an initial object 0 and a coproduct $A + B$. Then the following diagram commutes.



If this diagram is a pullback (i.e. it is a universal commutative cone, see definition F.15) and the canonical injections I_A and I_B are monomorphisms, then the coproduct $A + B$ is a *disjoint coproduct*.

In several interesting categories (e.g. **Set**) monomorphisms are *complementable*:

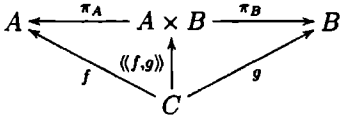
Definition F.12

An arrow $f : A \to B$ is *complementable* iff a $g : C \to B$ exists such that B is isomorphic with $A + C$ with f and g as the sum injection arrows. In this case g is a *complement* of f. The object C is frequently denoted as $B - A$.

The dual notion of coproduct is *product*. In the category **Set** a product corresponds to the notion of a cartesian product with associated projection functions.

Definition F.13

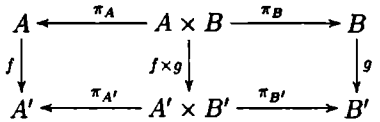
A *product* of two objects A and B in a category consists of an object $A \times B$ together with arrows $\pi_A : A \times B \to A$ and $\pi_B : A \times B \to B$ such that for any arrows $f : C \to A$ and $g : C \to B$, there is a unique arrow, denoted as $\langle\langle f, g \rangle\rangle : C \to A \times B$, such that the following diagram commutes:



As with coproducts, this definition can be straightforwardly extended to arrows.

Definition F.14

A *product* of two arrows $f : A \to A'$ and $g : B \to B'$ is an arrow $f \times g : A \times B \to A' \times B'$ such that the following diagram commutes:



F.4 Limits and colimits

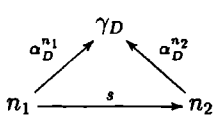
Limits and colimits are dual notions. Both concepts are very general and often used in category theory.

A *limit* is the categorical version of the concept of an equationally defined subset of a product. A product, therefore, is a special kind of limit. A *colimit* is the categorical version of a quotient of a sum by an equivalence relation. A coproduct, therefore, is a special kind of colimit. Only the definition of a colimit is given as the general notion of limit is not important in the context of this thesis.

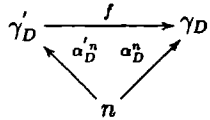
Definition F.15

Let G be a graph and C be a category. Let $D : \mathsf{G} \to \mathsf{C}$ be a diagram in C with shape G. A *cocone* with *base* D is an object γ_D (*apex*) together with a family $\left\{\alpha_D^n\right\}$ of arrows of C indexed by the nodes of G, such that $\alpha_D^n : n \to \gamma_D$ for each node of G. The arrow α_D^n is the *component* of the cocone at n. The cocone is written as $\left\{\alpha_D^n\right\} : D \to \gamma_D$, or simply $\alpha_D : D \to \gamma_D$.

The cocone is *commutative* if for any arrow $s : n_1 \to n_2$ of G, the following diagram commutes.



If $\alpha'_D : D \to \gamma'_D$ and $\alpha_D : D \to \gamma_D$ are cocones, an *arrow* from the first to the second is an arrow $f : \gamma'_D \to \gamma_D$ such that for each node n of G, the following diagram commutes.



A commutative cocone with base D is called *universal* if it has a unique arrow to every other commutative cocone with the same base. A universal cocone, if such exists, is called a *colimit* of the diagram D.

Appendix G

References to Songs

Well after all is said and done
Gotta move while it's still fun
But let me walk before they make me run

From: *"Before They Make Me Run",*
The Rolling Stones

This appendix contains references to the albums or singles of *The Rolling Stones*[1] song texts used at the beginning of each chapter and appendix.

[1]The web-site of The Rolling Stones is http.//www stones com.

Song	*Album/Single*	*Released*
Good Times, Bad Times	Single (B-side)	June, 1964
Start Me Up	Tattoo You	August, 1981
All Down The Line	Exile On Main Street	May, 1972
You Can't Always Get What You Want	Let It Bleed	December, 1969
Gimme Shelter	Let It Bleed	December, 1969
19th Nervous Breakdown	Single (B-side)	February, 1966
It's Only Rock 'n' Roll	It's Only Rock 'n' Roll	October, 1974
Jigsaw Puzzle	Beggars Banquet	December, 1968
Goin' Home	Aftermath	April, 1966
In Another Land	Their Satanic Majesties Request	December, 1967
Fingerprint File	It's Only Rock 'n' Roll	October, 1974
Hide Your Love	Goats Head Soup	August, 1973
Hot Stuff	Black And Blue	April, 1976
Sway	Sticky Fingers	April, 1971
Send It To Me	Emotional Rescue	June, 1980
Before They Make Me Run	Some Girls	June, 1978
You Got Me Rocking	Voodoo Lounge	July, 1994
I'm Going Down	Metamorphosis	June, 1975
I'm Free	Out Of Our Heads	September, 1965
Paint It Black	Single (A-side)	May, 1966
Sympathy For The Devil	Beggars Banquet	December, 1968

Bibliography

Hey hey, you got me rocking now
Hey hey, you got me rocking now

From: "You Got Me Rocking",
The Rolling Stones

[ADM+89] M. Atkinson, D. DeWitt, D. Maier, F. Bancilhon, K.R. Dittrich, and S.B. Zdonik. The object-oriented database system manifesto. In W. Kim, J.-M. Nicolas, and S. Nishio, editors, *Proceedings of the First International Conference on Deductive and Object-Oriented Databases (DOOD-89)*, pages 40–57, Kyoto, Japan, 1989. Elsevier Science Publishers.

[Adr92] P.W. Adriaans. *Language Learning from a Categorial Perspective.* PhD thesis, University of Amsterdam, Amsterdam, The Netherlands, 1992.

[AF88] D.E. Avison and G. Fitzgerald. *Information Systems Development: Methodologies, Techniques and Tools.* Blackwell Scientific Publications, Oxford, United Kingdom, 1988.

[AH87] S. Abiteboul and R. Hull. IFO: A Formal Semantic Database Model. *ACM Transactions on Database Systems*, 12(4):525–565, December 1987.

[AHS90] J. Adámek, H. Herrlich, and G.E. Strecker. *Abstract and Concrete Categories.* Pure and applied mathematics. John Wiley and Sons, New York, New York, 1990.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Massachusetts, 1986.

[BB97] F.C. Berger and P. van Bommel. Augmenting a characterization network with semantical information. *Information Processing & Management*, 1997. (in press).

[BCD+95] E. Buchholz, H. Cyriaks, A. Düsterhöft, H. Mehlan, and B. Thalheim. Applying a Natural Language Dialogue Tool for Designing Databases. In *Proceedings*

of the First International Workshop on Applications of Natural Language to Databases (NLDB'95), pages 119–133, Versailles, France, June 1995.

[BCG95] B. Berg, M. Cline, and M. Girou. Lessons learned from the os/400 oo project. *Communications of the ACM*, 38(10):54–64, October 1995.

[BFW96] P. van Bommel, P.J.M. Frederiks, and Th.P. van der Weide. Object-Oriented Modeling based on Logbooks. *The Computer Journal*, 39(9), 1996. (in press).

[BGM85] A. Borgida, S. Greenspan, and J. Mylopoulos. Knowledge Representation as the Basis for Requirements Specifications. *IEEE Computer*, 18:82–91, April 1985.

[BH94] H. Bennis and T. Hoekstra. *Generative Grammatica.* Floris Publications, Dordrecht, The Netherlands, 1994. (In Dutch).

[Bom94] P. van Bommel. Experiences with EDO: an Evolutionary Database Optimizer. *Data & Knowledge Engineering*, 13:243–263, 1994.

[Bom95] P. van Bommel. *Database Optimization: An Evolutionary Approach.* PhD thesis, University of Nijmegen, Nijmegen, The Netherlands, 1995.

[Boo91] G. Booch. *Object-Oriented Design with Applications.* Benjamin Cummings, Redwood City, California, 1991.

[BR95a] J.F.M. Burg and R.P. van de Riet. COLOR-X: Linguistically-based Event Modeling: A General Approach to Dynamic Modeling. In J. Iivari, K. Lyytinen, and M. Rossi, editors, *The Proceedings of the Seventh International Conference on Advanced Information System Engineering*, Lecture Notes in Computer Science, pages 26–39, Jyvaskyla, Finland, 1995. Springer-Verlag.

[BR95b] J F.M. Burg and R.P. van de Riet. COLOR-X: Object Modeling profits from Linguistics. In *Proceedings of the KB&KS'95, the Second International Conference on Building and Sharing of Very Large-Scale Knowledge Bases*, Enschede, The Netherlands, 1995.

[BR96a] G. Booch and J. Rumbaugh. Unified Method for Object-Oriented Development: Documentation Set. Technical Report, Rational Software Corporation, 1996.

[BR96b] J.F.M. Burg and R.P. van de Riet. COLOR-X: Using Knowledge from WordNet for Conceptual Modeling. In C. Fellbaum, editor, *WordNet: An Electronic Lexical Database and Some of its Applications.* MIT Press, 1996.

[Bra83] R.J. Brachman. What IS-A Is and Isn't: An Analysis of Taxonomic Links in Semantics Networks *IEEE Computer*, 16(10):30–36, 1983.

[BRC93] J.F.M. Burg, R.P. van de Riet, and S.C. Chang. A data-dictionary as a lexicon: An application of linguistics in information systems. In B. Bhargava, T. Finin, and Y. Yesha, editors, *Proceedings of the 2nd International Conference on Information and Knowledge Management*, 1993.

[BSW94] K. Baclawski, D. Simovici, and W. White. A categorical approach to database semantics. *Mathematical Structures in Computer Science*, 4:147–183, 1994.

[Bub86] J.A. Bubenko. Information System Methodologies - A Research View. In T.W. Olle, H.G. Sol, and A.A. Verrijn-Stuart, editors, *Information Systems Design Methodologies: Improving the Practice*, pages 289–318. North-Holland, Amsterdam, The Netherlands, 1986.

[Bur96] J.F.M. Burg. *Linguistic Instruments In Requirements Engineering*. PhD thesis, Free University, Amsterdam, The Netherlands, 1996.

[BW90a] J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Cambridge University Press, Cambridge, United Kingdom, 1990.

[BW90b] M. Barr and C. Wells. *Category Theory for Computing Science*. Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

[BW91] J.N. Brinkkemper and G.M. Wijers. CASE Tools and Methodologies: a European Perspective. In T. Bergin, editor, *CASE: Issues and Trends for for the 1990s and Beyond Idea Group Publishing*, 1991.

[BW94] E. Bertino and H. Weigand. An approach to authorization modeling in object-oriented database systems. *Data & Knowledge Engineering*, 12:1–29, 1994.

[Car84] L. Cardelli. A Semantics of Multiple Inheritance. In *International Symposium on Semantics of Data types*, volume 173 of *Lecture Notes in Computer Science*, pages 51–67, Berlin, Germany, June 1984. Springer-Verlag.

[Che76] P.P. Chen. The Entity-Relationship Model: Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, March 1976.

[Cho65] N. Chomsky. *Aspects of the theory of syntax*. MIT Press, Cambridge, Massachusetts, 1965.

[CM96] P.D. Chatzoglou and L.A. Macaulay. Requirements capture and IS methodologies. *Information Systems Journal*, 6(2):209–225, April 1996.

[Cod70] E.F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, 1970.

[Cor96] J.-P. Corriveau. Traceability Process for Large OO Projects. *IEEE Computer*, 29(9):63–68, September 1996.

[CP96] P.N. Creasy and H.A. Proper. A Generic Model for 3-Dimensional Conceptual Modelling. *Data & Knowledge Engineering*, 20:119–161, October 1996.

[CSS94] J.F. Costa, A. Sernadas, and C. Sernadas. Object inheritance beyond subtyping. *Acta Informatica*, 31(1):5–26, January 1994.

[CW93] M.A. Collignon and Th.P. van der Weide. An Information Analysis Method Based on PSM. In G.M. Nijssen, editor, *Proceedings of NIAM-ISDM*. NIAM-GUIDE, September 1993.

[CY90] P. Coad and E. Yourdon. *Object-Oriented Analysis*. Yourdon Press, New York, New York, 1990.

[Dal92] H. Dalianis. A method for validating a conceptual model by natural language discourse generation. In P. Loucopoulos, editor, *Proceedings of the Fourth International Conference CAiSE'92 on Advanced Information Systems Engineering*, volume 593 of *Lecture Notes in Computer Science*, pages 425–444, Manchester, United Kingdom, 1992. Springer-Verlag.

[Dal95] H. Dalianis. Aggregation, Formal Specification and Natural Language Generation. In *Proceedings of the First International Workshop on Applications of Natural Language to Databases (NLDB'95)*, pages 135–149, Versailles, France, June 1995.

[Dat91] C.J. Date. *An Introduction to Data Base Systems*, volume 1. Addison-Wesley, Reading, Massachusetts, 5th edition, 1991.

[Dav90] A.M. Davis. *Software Requirements: Analysis & Specification*. Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

[De 96] O.M.F. De Troyer. A formalization of the Binary Object-Role Model based on logic. *Data & Knowledge Engineering*, 19:1–37, September 1996.

[DFW96] C.F. Derksen, P.J.M. Frederiks, and Th.P. van der Weide. Paraphrasing as a Technique to Support Object-Oriented Analysis. In R P. van de Riet, J.F.M. Burg, and A.J. van der Vos, editors, *Proceedings of the Second Workshop on Applications of Natural Language to Databases (NLDB'96)*, pages 28–39, Amsterdam, The Netherlands, June 1996.

[Dig89] F.P.M. Dignum. *A Language for Modelling Knowledge Bases*. PhD thesis, Free University, Amsterdam, The Netherlands, 1989.

[Dij76] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[Dik89] S.C. Dik. *The Theory of Functional Grammar. Part I: The Structure of the Clause*. Floris Publications, Dordrecht, The Netherlands, 1989.

[Dit90] K.R. Dittrich. Object-oriented database systems: The next miles of the marathon. *Information Systems*, 15(1):161–167, 1990.

[DJM92] C.N.G. Dampney, M.S.J. Johnson, and G.P. Monro. An Illustrated Mathematical Foundation for ERA. In C.M.I. Rattray and R.G. Clark, editors, *The Unified Computation Laboratory*, pages 77–83, Oxford University Press, 1992. The Institute of Mathematics and Its Applications.

[DK92] E. Dürr and J. van Katwijk. VDM++: a formal specification language for object-oriented designs. In *Proceedings of the 9th International Conference on Software Engineering*, pages 214–219, New York, New York, 1992. IEEE Computer Society Press.

[DKNZ92a] C. Dekkers, C.H.A. Koster, M.-J. Nederhof, and A. van Zwol. Manual for Grammar WorkBench Version 1.5. Technical Report 92-14, Department of Computer Science, University of Nijmegen, Nijmegen, The Netherlands, July 1992.

[DKNZ92b] C. Dekkers, C.H.A. Koster, M.-J. Nederhof, and A. van Zwol. The Grammar Workbench: A First Step towards Lingware Engineering. In *Proceedings of the second Twente Workshop on Language Technology, Memoranda Informatica 92-29*, pages 103–115, Enschede, The Netherlands, April 1992. University of Twente.

[DMP84] O.M.F. De Troyer, R. Meersman, and F. Ponsaert. RIDL User Guide. Research report, International Centre for Information Analysis Services, Control Data Belgium, Inc., Brussels, Belgium, 1984.

[DN66] O.J. Dahl and K. Nygaard. Simula – an algol-based simulation language. *Communications of the ACM*, 9:71–78, 1966.

[DO90] L. Dunn and M.E. Orlowska. A Natural Language Interpreter for the Construction of Conceptual Schemas. In B. Steinholz, A. Sølvberg, and L. Bergman, editors, *Proceedings of the Second Nordic Conference CAiSE'90 on Advanced Information Systems Engineering*, volume 436 of *Lecture Notes in Computer Science*, pages 175–194, Stockholm, Sweden, 1990. Springer-Verlag.

[DS95] G. Dedene and M. Snoeck. Formal deadlock elimination in an object oriented conceptual schema. *Data & Knowledge Engineering*, 15:1–30, March 1995.

[EGH+92] G. Engels, M. Gogolla, U. Hohenstein, K. Hülsmann, P. Löhr-Richter, G. Saake, and H-D. Ehrich. Conceptual modelling of database applications using an extended ER model. *Data & Knowledge Engineering*, 9(4):157–204, 1992.

[EJW95] D.W. Embley, R.B. Jackson, and S.N. Woodfield. OO Systems Analysis: Is It or Isn't It? *IEEE Software*, 12(3):19-33, July 1995.

[ES91] H.-D. Ehrich and A. Sernadas. Object concepts and constructions. In G. Saake and A. Sernadas, editors, *Proceedings of the IS-CORE Workshop'91 (Informatik-Berichte 91-03)*, pages 1–24, Braunschweig, Germany, 1991. Technische Universität Braunschweig.

[Fah91] C. Fahner. *Taal voor Welzijn.* De Vijverberg, Kampen, The Netherlands, 1991. (In Dutch).

[FC96] M.E. Fayad and M. Cline. Managing Object-Oriented Software Development. *IEEE Computer*, 29(9):26–31, September 1996.

[FHL97] P.J.M. Frederiks, A.H.M. ter Hofstede, and E. Lippe. A unifying framework for conceptual data modelling concepts. *Information and Software Technology*, 39(1):15–25, January 1997.

[Fit96] B. Fitzgerald. Formalized systems development methodologies: a critical perspective. *Information Systems Journal*, 6(1):3–23, January 1996.

[FKW95] P.J.M. Frederiks, C.H.A. Koster, and Th.P. van der Weide. Object-Oriented Analysis using Informal Language. Technical Report CSI-R9516, Computing Science Institute, University of Nijmegen, Nijmegen, The Netherlands, December 1995.

[FKW96] P.J.M. Frederiks, C.H.A. Koster, and Th.P. van der Weide. Validation of Object-Oriented Analysis Models using Informal Language. Technical Report CSI-R9609, Computing Science Institute, University of Nijmegen, Nijmegen, The Netherlands, May 1996.

[FSMS91] J. Fiadeiro, C. Sernadas, T. Maibaum, and G. Saake. Proof-theoretic semantics of object-oriented specification constructs. In R. Meersman, W. Kent, and S. Khosla, editors, *Object-oriented databases: analysis, design and construction*, pages 243–284, Amsterdam, The Netherlands, 1991. North-Holland.

[FW96a] P.J.M. Frederiks and Th.P. van der Weide. A Note on Valid Instance Categories for Conceptual Data Modeling. Technical Note CSI-N9610, December 1996.

[FW96b] P.J.M. Frederiks and Th.P. van der Weide. Cognitive Requirements for Natural Language Based Conceptual Modeling. Technical Report CSI-R9610, Computing Science Institute, University of Nijmegen, Nijmegen, The Netherlands, June 1996.

[FW96c] P.J.M. Frederiks and Th.P. van der Weide. Formalization, Integration, and Validation of Object-Oriented Analysis Models leading to an Information Grammar. Technical Report CSI-R9625, Computing Science Institute, University of Nijmegen, Nijmegen, The Netherlands, December 1996.

[FW96d] P.J.M. Frederiks and Th.P. van der Weide. From a File-Oriented View to an Object-Oriented View. Technical Report CSI-R9601, Computing Science Institute, University of Nijmegen, Nijmegen, The Netherlands, January 1996.

[FW96e] P.J.M. Frederiks and Th.P. van der Weide. Verification and Design for Information Architectures. Technical Note CSI-N9611, December 1996.

[Gle91] H. Gleitman. *Psychology.* W.W. Norton and Compagny, New York, New York, 1991.

[Gog91] J.A. Goguen. A categorical manifesto. *Mathematical Structures in Computer Science*, 1(1):49–67, 1991.

[Gra94] I. Graham. *Object-oriented Methods.* Addison-Wesley, Reading, Massachusetts, 1994.

[Gri82] J.J. van Griethuysen, editor. *Concepts and Terminology for the Conceptual Schema and the Information Base.* Publ. nr. ISO/TC97/SC5-N695, 1982.

[Hal95] T.A. Halpin. *Conceptual Schema and Relational Database Design.* Prentice-Hall, Sydney, Australia, 2nd edition, 1995.

[Hat96] L. Hatton. Is modularization always a good idea? *Information and Software Technology*, 38(11):719–721, 1996.

[HBW95] A.H.M. ter Hofstede, F J.M. Bosman, and Th.P. van der Weide. Toward a Data Modeling Shell. Technical Report CSI-R9513, Computing Science Institute, University of Nijmegen, Nijmegen, The Netherlands, November 1995.

[HE92] U. Hohenstein and G. Engels. SQL/EER-syntax and semantics of an entity-relationship-based query Language. *Information Systems*, 17(3):209–242, 1992.

[HH97] J.W.G.M. Hubbers and A.H.M. ter Hofstede. An Investigation of the Formal Foundations of Object-Oriented Conceptual Data Modeling. Technical report, Computing Science Institute, University of Nijmegen, Nijmegen, The Netherlands, 1997. (to appear).

[HK87] R. Hull and R. King. Semantic Database Modelling: Survey, Applications and Research Issues. *ACM Computing Surveys*, 19(3):201–260, September 1987.

[HLF96] A.H.M. ter Hofstede, E. Lippe, and P.J.M. Frederiks. Conceptual Data Modeling from a Categorical Perspective. *The Computer Journal*, 39(3):215–231, August 1996.

[HLW97] A.H.M. ter Hofstede, E. Lippe, and Th.P. van der Weide. A categorical framework for conceptual data modeling: Definition, application, and implementation. *Acta Informatica*, 1997. (to apear).

[HM81] M. Hammer and D. McLeod. Database Description with SDM: A Semantic Database Model. *ACM Transactions on Database Systems*, 6(3):351–386, September 1981.

[HO92] T.A. Halpin and M.E. Orlowska. Fact-oriented modelling for data analysis. *Journal of Information Systems*, 2(2):97–119, April 1992.

[Hoa89] C.A.R. Hoare. Notes on an Approach to Category Theory for Computer Scientists. In M. Broy, editor, *Constructive Methods in Computing Science*, volume 55 of *NATO Advanced Science Institute Series*, pages 245–305. Springer-Verlag, 1989.

[Hof93] A.H.M. ter Hofstede. *Information Modelling in Data Intensive Domains.* PhD thesis, University of Nijmegen, Nijmegen, The Netherlands, 1993.

[HPW93] A.H.M. ter Hofstede, H.A. Proper, and Th.P. van der Weide. Formal definition of a conceptual language for the description and manipulation of information models. *Information Systems*, 18(7):489–523, October 1993.

[HPW94] A.H.M. ter Hofstede, H.A. Proper, and Th.P. van der Weide. Grammar Based Information Modelling. Technical Report CSI-R9414, Computing Science Institute, University of Nijmegen, Nijmegen, The Netherlands, October 1994.

[HSB+96] E. Hoenkamp, L. Schomaker, P. van Bommel, C.H.A. Koster, and Th.P. van der Weide. Profile - A Proactive Information Filter. Technical Note CSI-N9602, Computing Science Institute, University of Nijmegen, Nijmegen, The Netherlands, February 1996.

[HV96] A.H.M. ter Hofstede and T.F. Verhoef. Meta-CASE: Is the Game worth the Candle? *Information Systems Journal*, 6(1):41–68, 1996.

[HW93] A.H.M. ter Hofstede and Th.P. van der Weide. Expressiveness in conceptual data modelling. *Data & Knowledge Engineering*, 10(1):65–100, February 1993.

[HW94] A.H.M. ter Hofstede and Th.P. van der Weide. Fact Orientation in Complex Object Role Modelling Techniques. In T.A. Halpin and R. Meersman, editors, *Proceedings of the First International Conference on Object-Role Modelling (ORM-1)*, pages 45–59, Townsville, Australia, July 1994.

[IP94] A. Islam and W. Phoa. Category Models of Relational Databases I: Fibrational Formulation, Schema Integration. In M. Hagiya and J.C. Mitchell, editors, *Theoretical Aspects of Computer Software, International Symposium TACS'94*, volume 789 of *Lecture Notes in Computer Science*, pages 618–641, Sendai, Japan, April 1994. Springer-Verlag.

[Jac95] M.A. Jackson. *Software Requirements and Specifications.* Addison-Wesley, Reading, Massachusetts, 1995.

[Jac96] D. Jackson. Requirements Need Form, Maybe Formality. *IEEE Software*, 12(6):21–22, March 1996.

[JCJO92] I. Jacobson, M. Christerson, M. Jonsson, and P. van Overgaard. *OO Software Engineering, A Use Case Driven Approach.* Addison-Wesley, Reading, Massachusets, 1992.

[Joh95] P. Johannesson. Supporting Schema Integration by Linguistic Instruments. In *Proceedings of the First International Workshop on Applications of Natural Language to Databases (NLDB'95)*, pages 41–56, Versailles, France, June 1995.

[JSHS96] R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas. TROLL – A Language for Object-Oriented Specification of Information Systems. *ACM Transactions on Information Systems*, 14(2):175–211, 1996.

[KB96] G. Kovács and P. van Bommel. From Conceptual Model to OO Database via Intermediate Specification. Technical Report CSI-R9617, Computing Science Institute, University of Nijmegen, Nijmegen, The Netherlands, October 1996.

[Ken84] F. Kensing. Towards Evaluation of Methods for Property Determination: A Framework and a Critique of the Yourdon-DeMarco Approach. In Th.M.A. Bemelmans, editor, *Beyond Productivity: Information Systems Development for Organizational Effectiveness*, pages 325–338. North-Holland, Amsterdam, The Netherlands, 1984.

[Kin89] R. King. My Cat is Object-Oriented. In W. Kim and F.H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, ACM Press, Frontier Series, pages 23–30. Addison-Wesley, Reading, Massachusetts, 1989.

[KL89] W. Kim and F.H. Lochovsky. *Object-Oriented Concepts, Databases, and Applications.* ACM Press, Frontier Series. Addison-Wesley, Reading, Massachusetts, 1989.

[KM90] T. Korson and J. McGregor. Understanding Object Oriented: A Unifying Paradigm. *Communications of the ACM*, 33(9):40–60, September 1990.

[KO96] C.H.A. Koster and E. Oltmans. Proceedings of the first AGFL Workshop. Technical Report CSI-R9604, Computing Science Institute, University of Nijmegen, Nijmegen, The Netherlands, January 1996.

[Kos70] C.H.A. Koster. Affix grammars. In *ALGOL 68 implementation, Proceedings of the IFIP Working Conference on ALGOL 68 Implementation 1970*, pages 95–109, Amsterdam, The Netherlands, 1970. North-Holland.

[Kos91] C.H.A. Koster. Affix Grammars for Natural Languages. In H. Alblas and B. Melichar, editors, *Attribute Grammars, Applications and Systems, International Summer School SAGA*, volume 545 of *Lecture Notes in Computer Science*, pages 469–484. Springer-Verlag, Berlin, Germany, June 1991.

[Kos95a] C.H.A. Koster. Object-Oriented Design. Lecture Notes, University of Nijmegen, 1995.

[Kos95b] C.H.A. Koster. Personal Communication, June 1995.

[Kos96] C.H.A. Koster. AGFL Grammars for full-text Information Retrieval. Technical Report CSI-R9605, Department of Computer Science, University of Nijmegen, Nijmegen, The Netherlands, February 1996.

[Kos97] C.H.A. Koster. Personal Communication, February 1997.

[Kri94] G. Kristen. *Object Orientation, the KISS Method: From Information Architecture to Information System.* Addison-Wesley, Reading, Massachusetts, 1994.

[Law96] B. Lawrence. Do You Really Need Formal Requirements? *IEEE Software*, 12(6):20–22, March 1996.

[Lek93] H. van der Lek. On the Structure of an Information Grammar. In G.M. Nijssen, editor, *Proceedings of NIAM-ISDM.* NIAM-GUIDE, September 1993.

[LH96] E. Lippe and A.H.M. ter Hofstede. A Category Theory Approach to Conceptual Data Modeling. *RAIRO Theoretical Informatics and Applications*, 30(1):31–79, 1996.

[Mai88] D. Maier. *The Theory of Relational Databases.* Computer Science Press, Rockville, Maryland, 1988.

[Mat81] L. Mathiassen. *Systemudvikling og Systemudviklings-Metode.* PhD thesis, Aarhus University, Aarhus, Denmark, 1981. (In Danish).

[MBF+90] G.A. Miller, R. Beckwith, C. Fellbaum, D. Gross, and K.J. Miller. Introduction to wordnet: An on-line lexical database. *Journal of Lexicography*, 3(4):234–244, 1990.

[MBF+93] G.A. Miller, R. Beckwith, C. Fellbaum, D. Gross, and K.J. Miller. Five Papers on WordNet. Technical Report, Cognitive Science Laboratory, Princeton University, 1993.

[McC89] C.L. McClure. *CASE is Software Automation.* Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

[McL92] C. McLarty. *Elementary Categories, Elementary Toposes*, volume 21 of *Oxford Logic Guides.* Clarendon Press, Oxford, United Kingdom, 1992.

[Mee78] L.G.L.T. Meertens. Program text and program structure. In P.G. Hibbard and S.A. Schuman, editors, *Constructing quality software*, pages 271–283, Amsterdam, The Netherlands, May 1978. North-Holland. IFIP WG 2.1/WG 2.4 Working Conference, Novosibirsk.

[Mee82] R. Meersman. The RIDL Conceptual Language. Research report, International Centre for Information Analysis Services, Control Data Belgium, Inc., Brussels, Belgium, 1982.

[Mey88] B. Meyer. *Object-oriented software construction.* Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[Mey96] B. Meyer. Gurus share insights on objects. *Computer*, pages 95–98, June 1996.

[MG94] L. Mich and R. Garigliano. A Linguistic Approach to the Development of Object Oriented Systems using the NL System LOLITA. In E. Bertino and S. Urban, editors, *Proceedings of the International Symposium, ISOOMS '94: Object-Oriented Methodologies and Systems*, volume 858 of *Lecture Notes in Computer Science*, pages 371–386, Palermo, Italy, September 1994. Springer-Verlag.

[Mil91] G.A. Miller. *The science of words.* Scientific American Library, New York, New York, 1991.

[MN96] S. Moser and O. Nierstrasz. The Effect of Object-Oriented Frameworks on Developer Productivity. *IEEE Computer*, 29(9):45–51, September 1996.

[MO92] J. Martin and J. Odell. *Object-Oriented Analysis & Design.* Prentice-Hall, Englewood Cliffs, New Jersey, 1992.

[Ned94] M.-J. Nederhof. *Linguistic Parsing and Program Transformations.* PhD thesis, University of Nijmegen, Nijmegen, The Netherlands, 1994.

[NF84] G.M. Nijssen and E.D. Falkenberg. *Introduction to IBM SQL, Release 2.* Nijssen Data Bases Pty. Ltd., Chapel Hill, Queensland, Australia, 1984.

[NH89] G.M. Nijssen and T.A. Halpin. *Conceptual Schema and Relational Database Design: a fact oriented approach.* Prentice-Hall, Sydney, Australia, 1989.

[Nij89] G.M. Nijssen. An Axiom and Architecture for Information Systems. In E. D. Falkenberg and P. Lindgreen, editors, *Information System Concepts: An In-depth Analysis*, pages 157–175. North-Holland/IFIP, Amsterdam, The Netherlands, 1989.

[NR89] G.T. Nguyen and D. Rieu. Schema evolution in object-oriented database systems. *Data & Knowledge Engineering*, 4:43–67, 1989.

[Par90] H. Partsch. *Specification and Transformation of Programs - a Formal Approach to Software Development.* Springer-Verlag, Berlin, Germany, 1990.

[PG96] F.A.C. Pinheiro and J.A. Goguen. An Object-Oriented Tool for Tracing Requirements. *IEEE Software*, 12(6):52–64, March 1996.

[PP94] G. Pomberger and W. Pree. Quantitative and Qualitative Aspects of Object-Oriented Software Development. In E. Bertino and S. Urban, editors, *Proceedings of the International Symposium, ISOOMS '94: Object-Oriented Methodologies and Systems*, volume 858 of *Lecture Notes in Computer Science*, pages 96–107, Palermo, Italy, September 1994. Springer-Verlag.

[Pro94] H.A. Proper. *A Theory for Conceptual Modelling of Evolving Application Domains.* PhD thesis, University of Nijmegen, Nijmegen, The Netherlands, 1994.

[PSB+94] J. Preece, H. Sharp, D. Benyon, S. Holland, and T. Carey. *Human Computer Interaction.* Addison-Wesley, Reading, Massachusetts, 1994.

[PW95a] H.A. Proper and Th.P. van der Weide. A General Theory for the Evolution of Application Models. *IEEE Transactions on Knowledge and Data Engineering*, 7(6):984–996, December 1995.

[PW95b] H.A. Proper and Th.P. van der Weide. Information Disclosure in Evolving Information Systems: Taking a shot at a moving target. *Data & Knowledge Engineering*, 15:135–168, 1995.

[Qui60] W. Quine. *Word and object - Studies in communication.* The Technology Press of the Massachusetts Institute of Technology, Cambridge, Massachusetts, 1960.

[RB88] D.E. Rydeheard and R.M. Burstall. *Computational Category Theory.* Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorenson. *Object-Oriented Modeling and Design.* Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[RDPS96] A. Ruiz-Delgado, D. Pitt, and C. Smythe. A Review of Object-oriented Approaches in Formal Methods. *The Computer Journal*, 38(10):777–784, 1996.

[Rie94] R.P. van de Riet. Linguistic Instruments in Knowledge Engineering, A Research Proposal and some Experiments. In *Proceedings of the Knowledge Building and Knowledge Sharing (KBKS'94)*, pages 200–207, Tokyo, Japan, 1994. IOS Press.

[Rij75] C.J. van Rijsbergen. *Information Retrieval.* Butterworths, London, United Kingdom, 1975.

[RP92] C. Rolland and C. Proix. A Natural Language Approach For Requirements Engineering. In P. Loucopoulos, editor, *Proceedings of the Fourth International Conference CAiSE'92 on Advanced Information Systems Engineering*, volume 593 of *Lecture Notes in Computer Science*, pages 257–277, Manchester, United Kingdom, 1992. Springer-Verlag.

[RP96] D. Redmond-Pyle. Software development methods and tools: some trends and issues. *Software Engineering Journal*, 11(2):99–103, March 1996.

[RS96] R. Richardson and A.F. Smeaton. An Information Retrieval Approach to Locating Information in Large Scale Federated Database Systems. In R.P. van de Riet, J.F.M. Burg, and A.J. van der Vos, editors, *Proceedings of the Second Workshop on Applications of Natural Language to Databases (NLDB'96)*, pages 52–64, Amsterdam, The Netherlands, June 1996.

[SBC92] S. Stepney, R. Barden, and D. Cooper. *Object Orientation in Z.* Workshops in Computing. Springer-Verlag, Berlin, Germany, 1992.

[SBF96] S. Sparks, K. Benner, and C. Faris. Managing Object-Oriented Framework Reuse. *IEEE Computer*, 29(9):52–61, September 1996.

[Sch92] B. Schneiderman. *Designing the User Interface: Strategies for Effective Computer Interaction.* Addison-Wesley, Reading, Massachusetts, 2nd edition, 1992.

[SD96a] B. Sadr and P.J. Dousette. An OO Project Management Strategy. *IEEE Computer*, 29(9):33–38, September 1996.

[SD96b] M. Snoeck and G. Dedene. Generalization/specialization and role in object oriented conceptual modeling. *Data & Knowledge Engineering*, 19:171–195, September 1996.

[SFMS89] C. Sernadas, J. Fiadeiro, R. Meersman, and A. Sernadas. Proof-theoretic Conceptual Modelling: the NIAM Case Study. In E.D. Falkenberg and P. Lindgreen, editors, *Information System Concepts: An In-depth Analysis*, pages 1–30, Amsterdam, The Netherlands, 1989. North-Holland/IFIP.

[Shi81] D.W. Shipman. The Functional Data Model and the Data Language DAPLEX. *ACM Transactions on Database Systems*, 6(1):140–173, March 1981.

[Sie90] A. Siebes. *On Complex Objects.* PhD thesis, University of Twente, Enschede, The Netherlands, 1990.

[Sme92] A.F. Smeaton. Progress in the Application of Natural Language Processing to Information Retrieval Tasks. *The Computer Journal*, 35, June 1992.

[Sno90] R. Snodgrass. Temporal Databases Status and Research Directions. *SIGMOD Record*, 19(4):83–89, December 1990.

[Sno95] M. Snoeck. *On a Process Algebra Approach for the Construction and Analysis of M.E.R.O.DE.-based Conceptual Models.* PhD thesis, University of Leuven, Leuven, Belgium, 1995.

[Sny93] A. Snyder. The Essence of Objects: Concepts and Terms. *IEEE Software*, 1(10):31–42, 1993.

[Sol83] H.G. Sol. A Feature Analysis of Information Systems Design Methodologies: Methodological Considerations. In T.W. Olle, H.G. Sol, and C.J. Tully, editors, *Information Systems Design Methodologies: A Feature Analysis*, pages 1–7. North-Holland/IFIP, Amsterdam, The Netherlands, 1983.

[Sol88] H.G. Sol. Information Systems Development: A Problem Solving Approach. In *Proceedings of 1988 INTEC Symposium Systems Analysis and Design: A Research Strategy*, Atlanta, Georgia, 1988.

[SS96] J. Siddiqi and M.C. Shekaran. Requirements Engineering: The Emerging Wisdom. *IEEE Software*, 12(6):15–19, March 1996.

[STR95] V.C. Storey, C.B. Thompson, and S. Ram. Understanding database design expertise. *Data & Knowledge Engineering*, 16(2):97–124, August 1995.

[TCF90] L. Tucherman, M.A. Casanova, and A.L. Furtado. The CHRIS consultant - A tool for database design and rapid prototyping. *Information Systems*, 15(2):187–195, 1990.

[TL91] C.I. Theodoulidis and P. Loucopoulos. The Time Dimension in Conceptual Modelling. *Information Systems*, 16(3):273–300, 1991.

[TLH+88] W.S. Turner, R.P. Langerhorst, G.F. Hice, H.B. Eilers, and A.A. Uijttenbroek. *SDM: System Development Methodology.* North-Holland, Amsterdam, The Netherlands, 1988.

[Tre91] M.T. Tresch. A Framework for Schema Evolution by Meta Object Manipulation. In *Proceedings of the 3d International Workshop on Foundations of Models and Languages for Data and Objects*, Aigen, Austria, September 1991. Institut für Informatik, TU Clausthal.

[Tui94] C. Tuijn. *Data Modeling from a Categorical Perspective.* PhD thesis, University of Antwerp, Antwerp, Belgium, 1994.

[TY96] J.Y.L. Thong and C.-S. Yap. Information Systems Effectiveness: A User Satisfaction Approach. *Information Processing and Management*, 32(5):601–610, 1996.

[TYF86] T.J. Teorey, D. Yang, and J.P. Fry. A logical design methodology for relational databases using the extended entity-relationship model. *ACM Computing Surveys*, 18(2):197–222, 1986.

[US83] Department of Defense US. Reference manual for the Ada programming language. ANSI/MIL-STD 1815 A, 1983.

[Vaa96] F.W. Vaandrager. De Ingebouwde Informatica. Computing Science Institute, University of Nijmegen, Nijmegen, The Netherlands, December 1996. Inaugurele rede (in dutch).

[VBFW96] E.S.C. van de Ven, P. van Bommel, P.J.M. Frederiks, and Th.P. van der Weide. Natural Language Properties and Information Systems. Technical Note CSI-N9608, October 1996.

[Ven96] E.S.C. van de Ven. A Study on the Profits of Semantics in Object-oriented Information Modelling based on Natural Lanuage – An Artificial Intelligence approach. Master's thesis, University of Nijmegen, June 1996. No. 371.

[VH95] T.F. Verhoef and A.H.M. ter Hofstede. Feasibility of Flexible Information Modelling Support. In J. Iivari, K. Lyytinen, and M. Rossi, editors, *Proceedings of the Seventh International Conference CAiSE'95 on Advanced Information Systems Engineering*, volume 932 of *Lecture Notes in Computer Science*, pages 168–185, Jyväskylä, Finland, June 1995. Springer-Verlag.

[VHW91] T.F. Verhoef, A.H.M. ter Hofstede, and G.M. Wijers. Structuring modelling knowledge for CASE shells. In R. Andersen, J.A. Bubenko, and A. Sølvberg, editors, *Proceedings of the Third International Conference CAiSE'91 on Advanced Information Systems Engineering*, volume 498 of *Lecture Notes in Computer Science*, pages 502–524, Trondheim, Norway, May 1991. Springer-Verlag.

[Voo94] E.M. Voorhees. Query Expansion Using Lexical-Semantic Relations. In *Proceedings of the Seventeenth Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval*, Dublin, Ireland, July 1994.

[Web94] S. Webster. An annotated bibliography for object-oriented analysis and design. *Information and Software Technology*, 36(9):569–582, 1994.

[WHB92] Th.P. van der Weide, A.H.M. ter Hofstede, and P. van Bommel. Uniquest: Determining the Semantics of Complex Uniqueness Constraints. *The Computer Journal*, 35(2):148–156, April 1992.

[Wie91] R. Wieringa. Steps towards a method for the formal modeling of dynamic objects. *Data & Knowledge Engineering*, 6:509 540, 1991.

[Wig95] T.A. Wiggerts. Using Worldviews in Conceptual Object Oriented Systems Modelling. Technical Report CSI-R9515, Computing Science Institute, University of Nijmegen, Nijmegen, The Netherlands, November 1995.

[Wij91] G.M. Wijers. *Modelling Support in Information Systems Development.* PhD thesis, Delft University of Technology, Delft, The Netherlands, 1991.

[Wil96a] J.D. Williams. Managing Iteration in OO Projects. *IEEE Computer*, 29(9):39–43, September 1996.

[Wil96b] C.P. Willis. Analysis of inheritance and multiple inheritance. *Software Engineering Journal*, 11:215-224, July 1996.

[Win90] J.J.V.R. Wintraecken. *The NIAM Information Analysis Method: Theory and Practice.* Kluwer, Deventer, The Netherlands, 1990.

[Wir71] N. Wirth. The programming language pascal. *Acta Informatica*, 1:35–63, 1971.

[Wir85] N. Wirth. *Programming in MODULA-2.* Springer-Verlag, Berlin, Germany, Third Edition, 1985.

[WMP+76] A. van Wijngaarden, B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, M. Sintzoff, C.H. Lindsey, L.G.L.T. Meertens, and R.G. Fisker. *Revised Report on the Algorithmic Language ALGOL 68.* Springer-Verlag, Berlin, Germany, 1976.

[WZ96] R. Weber and Y. Zhang. An analytical evaluation of NIAM's grammar for conceptual schema diagrams. *Information Systems Journal*, 6(2):147–170, April 1996.

[You89] E. Yourdon. *Modern Structured Analysis.* Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

[ZM90] S.B. Zdonik and D. Maier. *Readings in Object-Oriented Database Systems.* Morgan Kaufmann, San Mateo, California, 1990.

Index

Hey babe, what's wrong with you, girl
I have come around ringing your front door bell
Though our love fall down in dark clouds

From: "I'm Going Down",
The Rolling Stones

Author Index

I'm free to choose who I see any old time
I'm free to bring who I choose any old time

From: "I'm Free",
The Rolling Stones

Samenvatting

I see a red door and I want to paint it black
No colors anymore I want them to turn black

From: "Paint It Black",
The Rolling Stones

Met deze samenvatting hoop ik zowel collega's als mensen buiten het vakgebied Informatica een indruk te geven van het onderwerp en de resultaten van dit proefschrift.

Informatiesystemen

In onze samenleving is het gebruik van computergebaseerde systemen niet meer weg te denken. De meest voorkomende en gebruikte systemen zijn de zogenaamde *informatiesystemen*. Een informatiesysteem is een centaal gehouden systeem binnen een organisatie om *gegevens* op te slaan en te verwerken tot *informatie*, met als doel *informatiestromen* te versnellen en verbeteren.

Analoog aan disciplines als Bouwkunde wordt ook in de Informatica in de beginfase van de bouw van een informatiesysteem, de zogenaamde *analyse fase*, gebruik gemaakt van bouwtekeningen, *modellen* genaamd. Een speciaal soort modellen zijn de *conceptuele* modellen. Deze modellen proberen het probleem in kaart te brengen zonder daar aspecten van de oplossing in te verwerken. De conceptuele modellen tezamen met de gebruikte concepten en hun notatie worden aangeduid met de term *conceptuele modelleringstechniek*. Het proces van het opstellen van deze conceptuele modellen noemen we het *modelleringsproces*. Een modelleringstechniek is een onderdeel van wat men een *modelleringsmethode* noemt. Een modelleringsmethode bevat naast een modelleringstechniek bijvoorbeeld ook nog een werkwijze (de manier om tot de conceptuele modellen te komen), een wijze om het modelleringsproces te beheersen, en een wijze om het modelleringsproces te ondersteunen (al dan niet met automatische middelen).

Communicatie-gerichte informatiesystemen

De mensen binnen een organisatie hebben vaak een eigen bedrijfscultuur met een eigen taal, de zogenaamde *experttaal*. Voor een buitenstaander is die taal niet altijd te volgen. Net als mensen "spreken" informatiesystemen ook hun eigen taal. Deze taal is meestal heel anders

dan de experttaal. Voor simpele vragen aan het informatiesysteem kan de gebruiker dit verschil goed overbruggen. Naarmate de opdracht moeilijker wordt is meer expertise vereist. In het ideale geval hoeft een gebruiker niet de taal van het informatiesysteem te leren, maar leert het informatiesysteem de (expert)taal van de gebruiker. Zo'n informatiesysteem duiden we aan als een *communicatie-gericht* informatiesysteem.

Spreektalen, ook wel *natuurlijke talen* genoemd, en computertalen hebben hun eigen grammatica's. Bijvoorbeeld, de Nederlandse taal volgt de grammatica van het Nederlands. Ook de experttaal heeft een eigen grammatica met bijvoorbeeld bijzondere woorden en termen. Deze noemen we de *expertgrammatica*. De taal die het informatiesysteem "spreekt" volgt de *informatiegrammatica*. De taak van de ontwerpers van een communicatie-gericht informatiesysteem is dan ook om te zorgen dat de informatiegrammatica zoveel mogelijk zinnen uit de experttaal toelaat.

Dit is geen eenvoudige klus daar er aan (het gebruik van) natuurlijke taal, en dus ook de experttaal, eigenschappen kleven die zich niet eenvoudig laten automatiseren. Eén van die eigenschappen is de *ambiguïteit* van zinnen. Dat wil zeggen dat een zin voor meerdere uitleg vatbaar is. In ons dagelijks gebruik met taal hebben we daar niet zoveel last van. Als een zin niet wordt begrepen vragen we om toelichting, of we gebruiken gebaren ter verduidelijking. Een informatiesysteem is wat dat betreft een stugge communicatiepartner. Om dit soort problemen te ondervangen richten we ons op een versimpelde, meer gestructureerde vorm van de experttaal.

Flexibele informatiesystemen

Een organisatie is niet los te zien van zijn omgeving. Om goed te kunnen anticiperen op een voortdurend veranderende omgeving en op veranderingen binnen de organisatie is een flexibele houding en opzet van zowel de organisatie als zijn informatiesystemen gewenst. Dergelijke informatiesystemen duiden we aan als *flexibele* informatiesystemen.

Bij het ontwerpen van een informatiesysteem dient dus al rekening te worden gehouden met eventuele toekomstige veranderingen. Daartoe dient eerst de organisatie, waar het informatiesysteem een oplossing voor moet bieden, te worden geïnventariseerd. Als ook het daarbinnen op te lossen probleem eenmaal goed is begrepen kan er gewerkt worden aan een oplossing, rekening houdend met mogelijke toekomstige wijzigingen van de omgeving.

Al sinds de jaren zeventig zijn informatici op zoek naar modelleringsmethoden die geschikt zijn het steeds vaker veranderende en complexer wordende probleemgebied in kaart te brengen. Deze zoektocht vormt een deel van de *software crisis*. Alhoewel deze zoektocht nog niet ten einde is gekomen wordt er wel vooruitgang geboekt. De modelleringsmethoden en -technieken verbeteren, de opleidingen van informatici worden beter en er komt meer en betere automatische ondersteuning van het modelleringsproces.

Object-georiënteerd modelleren en de informatiegrammatica

In dit proefschrift wordt een modelleringsmethode geïntroduceerd voor het analyseren van informatiegrammatica's met als doel de bouw van flexibele en op communicatie-gerichte

informatiesystemen te ondersteunen.

Deze methode heeft in de eerste plaats een aantal kenmerken dat in het vakgebied *object-georiënteerd* wordt genoemd. Daarnaast gebruikt de methode natuurlijke taal als middel om de informatiegrammatica op te stellen en te controleren[2] en daarmee te komen tot een communicatie-gericht informatiesysteem. Een met natuurlijke taal beschreven *specificatie* van het probleemgebied vormt het uitgangspunt van de methode. Door het analyseren van deze *informele* specificatie proberen we de conceptuele modellen te verkrijgen. Deze *formele* modellen worden op hun beurt weer geverbaliseerd naar een beschrijving in natuurlijke taal. Op deze wijze is het voor de klant mogelijk de conceptuele modellen te begrijpen en valideren in zijn/haar eigen (natuurlijke) taal.

Het voert voor deze samenvatting te ver om te beschrijven wat object-oriëntatie nu precies inhoudt. Het is in de eerste plaats van belang op te merken dat men verwacht dat het gebruik van object-oriëntatie zal leiden tot flexibeler softwaresystemen. Daarnaast is er een aantal andere prettige kenmerken aan object-georiënteerde methoden. Zoals de naam object-oriëntatie al suggereert zijn methoden die op dit principe zijn gebaseerd gericht op het in kaart brengen van de objecten die optreden in het probleemgebied. Objecten kunnen gezien worden als (concrete en abstracte) dingen die we kunnen aanduiden. Object-georiënteerde conceptuele modellen brengen het probleemgebied in kaart waarbij gebruik gemaakt wordt van de terminologie van de experttaal. Dit heeft tot gevolg dat de *communicatie*[3] tussen analyst en klant (*domeindeskundige*) van het informatiesysteem in de terminologie van de klant kan verlopen hetgeen in het algemeen zal leiden tot een beter begrepen beschrijving van het probleem.

Tenslotte is er nog een aantal technische en commerciële voordelen aan het gebruik van object-oriëntatie. Bij het in kaart brengen van een organisatie zijn in het algemeen twee aspecten van belang: (1) wat zijn de relevante *gegevens* in deze organisatie en (2) welke *processen* spelen zich af binnen deze organisatie. Vaak werden/worden deze twee aspecten los van elkaar in kaart gebracht met verschillende modelleringstechnieken. Object-georiënteerde technieken ondersteunen het geïntegreerd in kaart brengen van zowel de relevante gegevens als de relevante processen.

Op object-georiënteerde wijze gebouwde softwaresystemen zijn goedkoper en beter te onderhouden (flexibiliteit). Daarnaast ondersteunt object-oriëntatie het ontwikkelen van op zichzelf staande softwaresystemen die voor meerdere probleemgebieden inzetbaar zijn. Met andere woorden, object-oriëntatie stimuleert hergebruik van software.

Het raamwerk van de methode

In het proefschrift van Dr. G.M. Wijers ([Wij91]) wordt een raamwerk voor modelleringsmethoden beschreven. Dit raamwerk (met enkele uitbreidingen daarop) heeft als uitgangspunt

[2]Het woord *controleren* is hier gebruikt, in de Nederlandse betekenis, ter vervanging van de jargonwoorden verifiëren en valideren en niet in de (Engelse) betekenis van beheersen en besturen ([Kos97]).

[3]In zijn onlangs uitgesproken inaugurele rede ([Vaa96]) benadrukt Prof. dr. F.W. Vaandrager het belang van communicatieve en sociale vaardigheden die een informaticus dient te hebben om zijn werk te kunnen uitvoeren. Daarnaast constateert Vaandrager dat de communicatieve aspecten bij de ontwikkeling van formele methoden voor de specificatie en analyse van software lang ondergewaardeerd zijn gebleven.

gediend voor het beschrijven van de methode die wordt behandeld in dit proefschrift. In het vervolg van deze samenvatting wordt dit raamwerk, en de invulling ervan, besproken.

Ten eerste moet er aan elke methode een *filosofie* ten grondslag liggen. In de methode van dit proefschrift is de filosofie gevolgd de prettige eigenschappen van natuurlijke taal zoveel mogelijk te benutten bij het opstellen en controleren van de conceptuele modellen.

Het gebruik van natuurlijke taal heeft ook als voordeel dat het modelleringsproces op een minder technisch niveau is te *beheersen* en te *sturen* voor diegenen die een softwareproject moeten leiden, de *projectleiders*. Een aantal van deze voordelen wordt toegelicht in dit proefschrift. Tevens worden er richtlijnen aan projectleiders verstrekt waaraan analisten en domeindeskundigen moeten voldoen om deze methode met succes te kunnen gebruiken.

Elke methode moet een *werkwijze* (stappenplan) hanteren om tot de gewenste resultaten te komen. Uiteraard moet deze werkwijze stroken met de filosofie van de methode. In dit proefschrift wordt een eerste opzet gemaakt voor een stappenplan waarin het gebruik van natuurlijk taal in de communicatie tussen analist en domeindeskundige wordt gestimuleerd. De hier gehanteerde werkwijze verlangt geen bovenmenselijke nauwkeurigheid, volledigheid en consistentie – deze eigenschappen worden iteratief bereikt.

Zoals eerder opgemerkt resulteert de analyse fase in een aantal conceptuele modellen. Deze modellen zijn op zichzelf weer op te vatten als een (wiskundige) taal. In dit proefschrift worden op uitgebreide en *formele* wijze de symbolen (de *syntax*) en de betekenis (de *semantiek*) van deze taal vastgelegd. Door deze formele aanpak is het mogelijk de modellen te *verifiëren*. Tevens wordt de relatie met de informatiegrammatica gelegd.

Om het werk van de analyst te ondersteunen en te controleren is het gewenst een methode uit te rusten met een aantal *automatische hulpmiddelen*. In dit proefschrift wordt gebruik gemaakt van een softwaresysteem dat in staat is op basis van de informatiegrammatica zinnen te genereren (zie bijvoorbeeld [DKNZ92b]). Daarnaast is dit systeem in staat te controleren of zinnen uit de experttaal ook daadwerkelijk zijn beschreven in de informatiegrammatica.

Het hierboven beschreven softwaresysteem, en de wijze waarop de informatiegrammatica kan worden verkregen uit de conceptuele modellen, stelt de analist en domeindeskundige in staat deze modellen te *valideren* in natuurlijke taal. Deze wijze van valideren kan er toe leiden dat mogelijke fouten in het model in een vroeg stadium worden gevonden.

Voorts is er een *tekenwijze* beschreven om de wiskundige modellen te *visualiseren*. In het verleden is gebleken dat het visualiseren van conceptuele modellen de communicatie (tussen analisten onderling) kan verbeteren en leidt tot een beter overzicht van het probleemgebied. Deze tekenwijze is beïnvloed door de tekenwijze van de conceptuele modelleringstechnieken NIAM[4] ([NH89]) en PSM[5] ([Hof93]).

Tot slot wil ik erop wijzen dat hier geen uitgerijpte methode wordt gepresenteerd, maar een gedetailleerde blauwdruk die laat zien hoe het ideaal, object-georiënteerd modelleren via een informatiegrammatica, naderbij kan worden gebracht.

[4]Natuurlijke taal Informatie Analyse Methode.
[5]Predicator Set Model.

Curriculum Vitæ

Pleased to meet you
hope you guess my name

From: "Sympathy For The Devil",
The Rolling Stones

Personalia

Name:	Paul Frederiks
Christian names:	Paulus Johannes Maria
Born:	April 30th 1968, Nijmegen, The Netherlands
Addresses:	University of Nijmegen Toernooiveld 1 NL-6525 ED Nijmegen The Netherlands
	Edmond R&D BV Van Trieststraat 1[a] NL-6512 CW Nijmegen The Netherlands
Email:	`paulf@edmond.nl` or `paulf@cs.kun.nl`
WWW:	http://www.cs.kun.nl/~paulf

Education & Jobs

1980 - 1984	VWO, Bisschop Bekkers College, Eindhoven
1984 - 1986	VWO, Bisschoppelijk College Schöndeln, Roermond
1986 - 1992	Computer Science, University of Nijmegen
1990 - June 1992	Teaching-assistant, University of Nijmegen
July 1992 - 1992	Scientific programmer, Programming Methodology, University of Nijmegen
1993 - 1996	PhD student, Information Systems, University of Nijmegen
1997 -	Software engineer, Edmond R&D BV

The End